Fractals

Parker Emmerson

November 2024

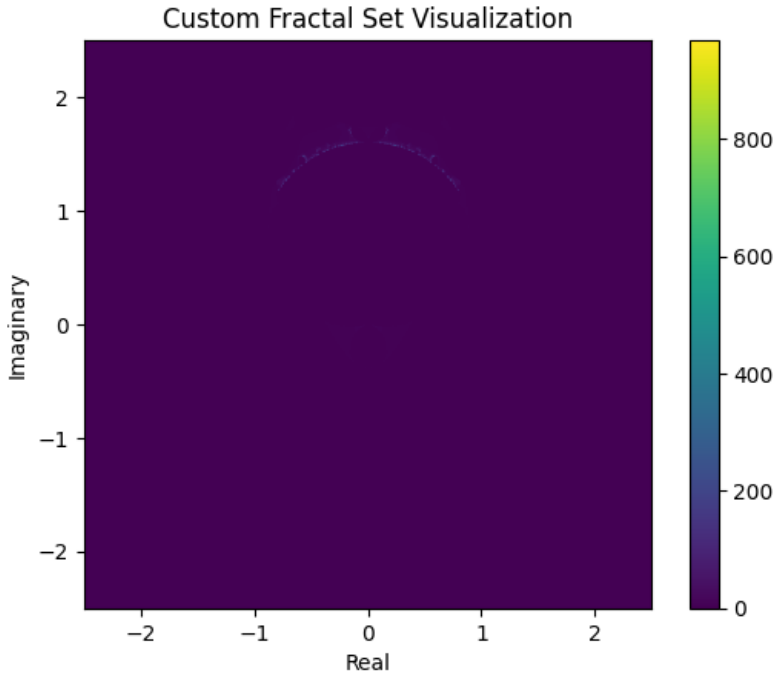
1 Introduction

These fractals are from interpretations of the following pseudo-code for the iterations:

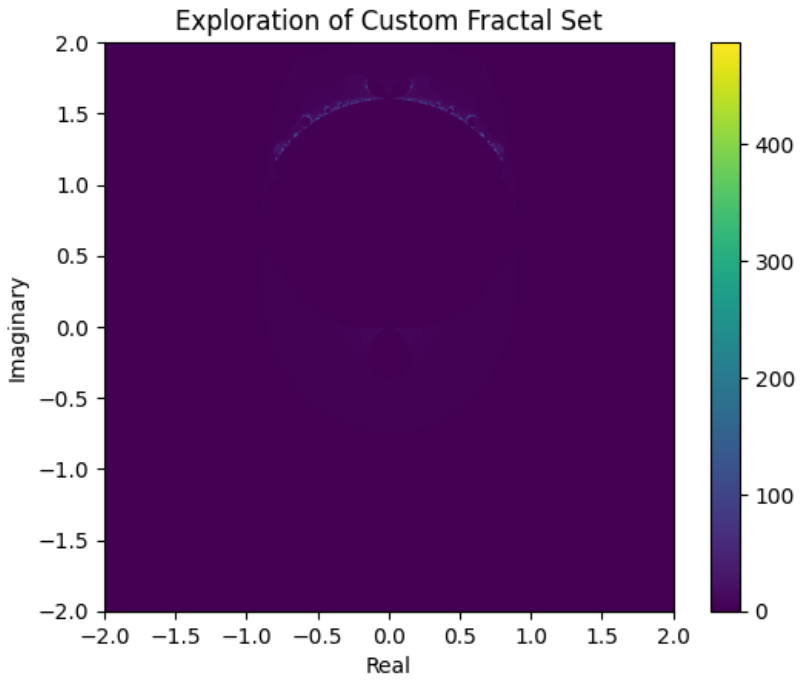
$$\theta r = \gamma x \pm jy$$
$$\theta_1 r_1 = \theta_2 r_2 - \theta_3 r_3 \ldots \theta_n r_n \ldots \theta_\infty r_\infty$$
$$\text{If } \theta_n r_n = 2\pi r_1 - 2\pi r_2, \text{ then}$$
$$\infty_{(2\pi \to \theta_n)} \neq \theta_\infty r_\infty \forall \theta_n r_n = f\left(\theta_2 r_2 - \theta_3 r_3\right)$$
$$\text{BECAUSE} \theta_1 r_1 = f\left(\sqrt{r_n - r_{(n-(n\pm 1))}}\right) * \theta_{(n-(n\pm 1))}$$
$$\therefore \text{ANY}_{(\infty \to \gamma)} = f\left(\theta_\infty r_\infty \ldots r_{(n-(n\pm 1))} \theta_{(n-(n\pm 1))} - \theta_1 r_1\right)$$
$$\leftarrow$$
$$f\left(\left(r_2\theta_2 - r_3\theta_3 \cdots\right) r_n \theta_n \cdots\right) r_\infty \theta_\infty$$

I then used a large language model to interpret the pseudo-code into a series of fractals. The following is the code and results.

2 Code and Visualizations



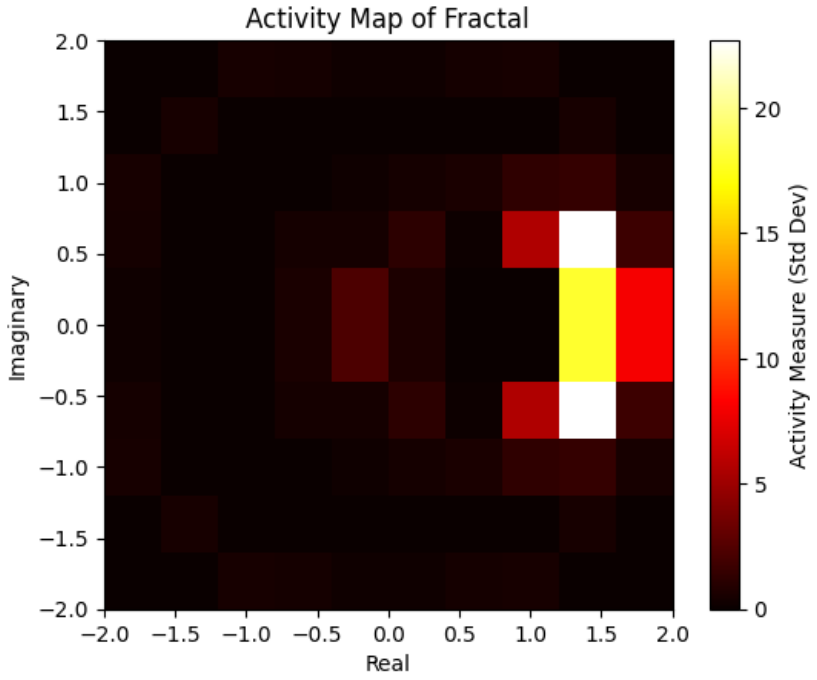
```python
import numpy as np
import matplotlib.pyplot as plt
from numba import jit

# Define a function to calculate the custom fractal set
@jit(nopython=True)
def compute_custom_fractal(real, imag, width, height, max_iter):
    fractal_set = np.zeros((height, width), dtype=np.int32)

    for i in range(height):
        for j in range(width):
            c = complex(real[j], imag[i])
            z = complex(0, 0)
            for k in range(max_iter):
                z = np.sin(z) - c * z + c  # Custom transformation
                if (z.real ** 2 + z.imag ** 2) >= 4:
                    fractal_set[i, j] = k
                    break

    return fractal_set

def generate_custom_fractal(n, x_min, x_max, y_min, y_max, max_iter):
    real = np.linspace(x_min, x_max, n)
    imag = np.linspace(y_min, y_max, n)

    fractal_set = compute_custom_fractal(real, imag, n, n, max_iter)

    plt.imshow(fractal_set.T, extent=[x_min, x_max, y_min, y_max], cmap='viridis', origin='lower')
    plt.colorbar()
    plt.title('Custom Fractal Set Visualization')
    plt.xlabel('Real')
    plt.ylabel('Imaginary')
    plt.show()

# Example of generating and visualizing the custom fractal set
generate_custom_fractal(1000, -2.5, 2.5, -2.5, 2.5, 1000)
```

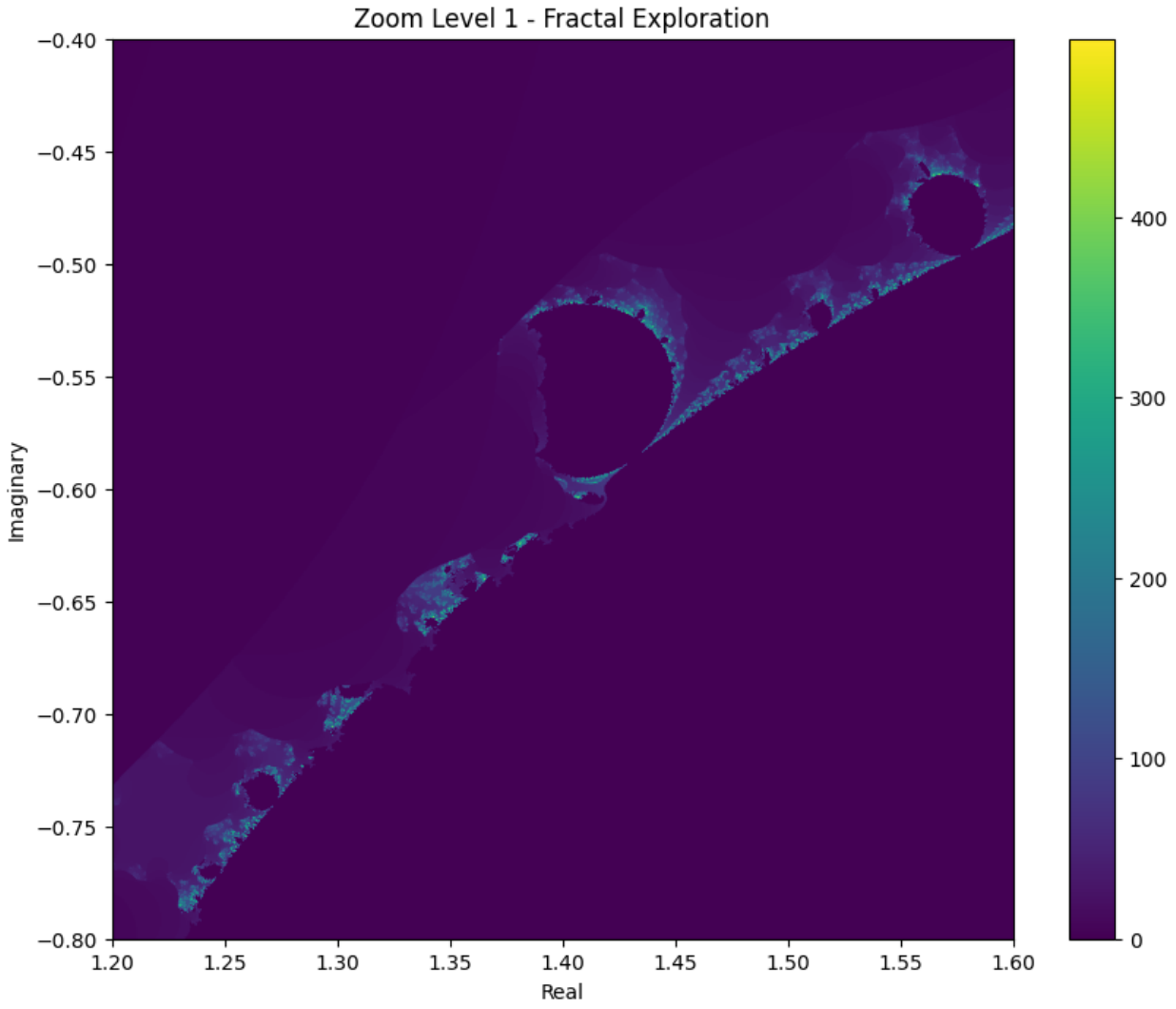


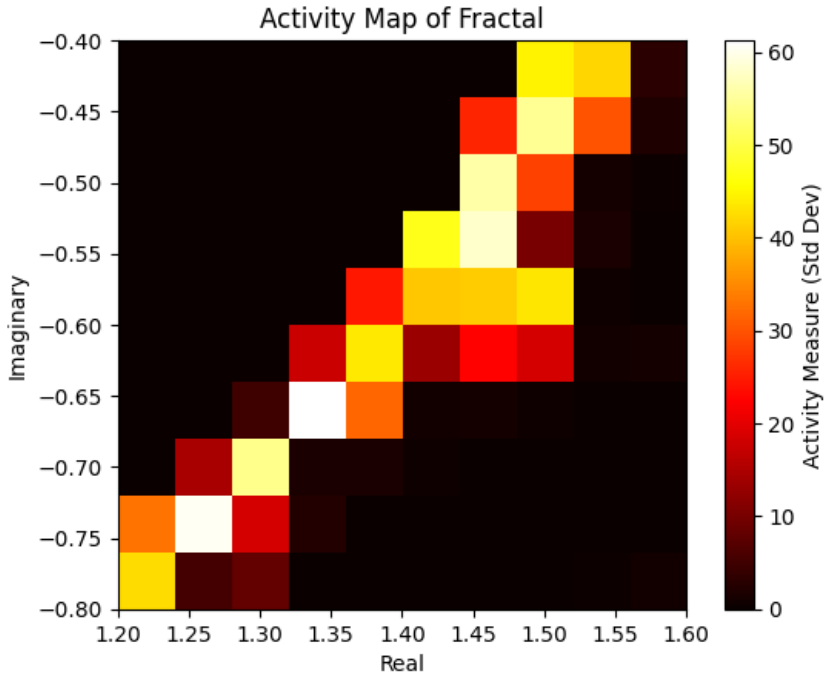

```python
import numpy as np
import matplotlib.pyplot as plt
from numba import jit

# Define the custom fractal function
@jit(nopython=True)
def compute_custom_fractal(real, imag, width, height, max_iter):
    fractal_set = np.zeros((height, width), dtype=np.int32)

    for i in range(height):
        for j in range(width):
            c = complex(real[j], imag[i])
            z = complex(0, 0)
            for k in range(max_iter):
                z = np.sin(z) - c * z + c  # Custom transformation
                if (z.real ** 2 + z.imag ** 2) >= 4:
                    fractal_set[i, j] = k
                    break

    return fractal_set

# Generate a fractal plot for a broad range
def explore_fractal(n, x_min, x_max, y_min, y_max, max_iter):
    real = np.linspace(x_min, x_max, n)
    imag = np.linspace(y_min, y_max, n)

    fractal_set = compute_custom_fractal(real, imag, n, n, max_iter)

    plt.imshow(fractal_set.T, extent=[x_min, x_max, y_min, y_max], cmap='viridis', origin='lower')
    plt.colorbar()
```

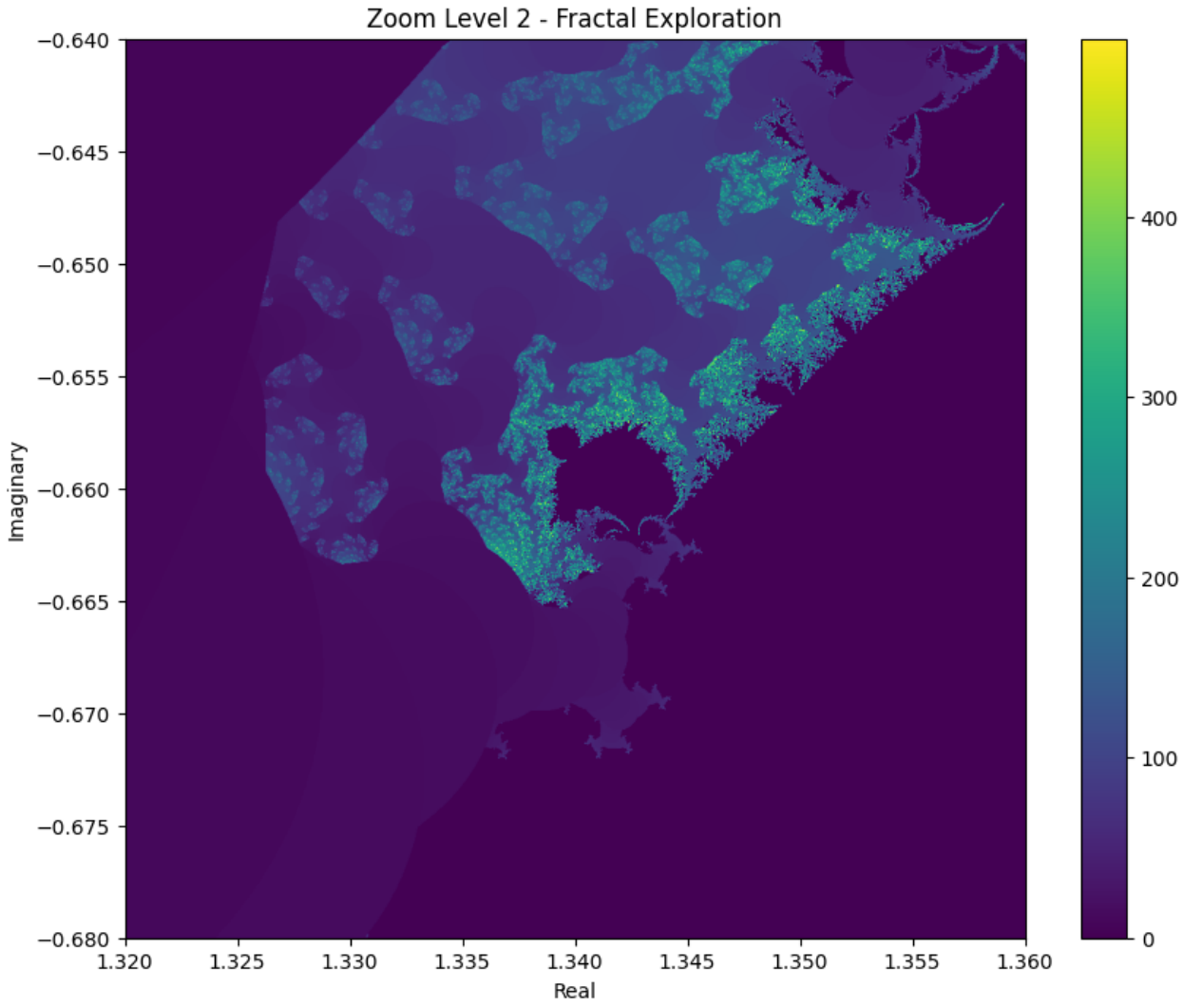
```
    plt.title('Exploration of Custom Fractal Set')
    plt.xlabel('Real')
    plt.ylabel('Imaginary')
    plt.show()

# Explore a large scale region
explore_fractal(1000, -2, 2, -2, 2, 500)
```

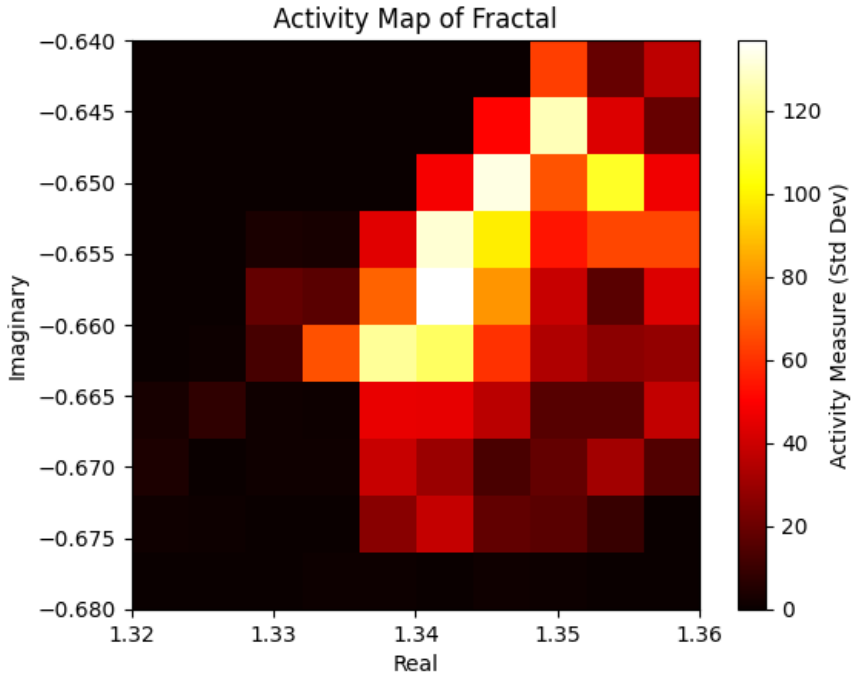
3 Zooming Functions and Activity Maps

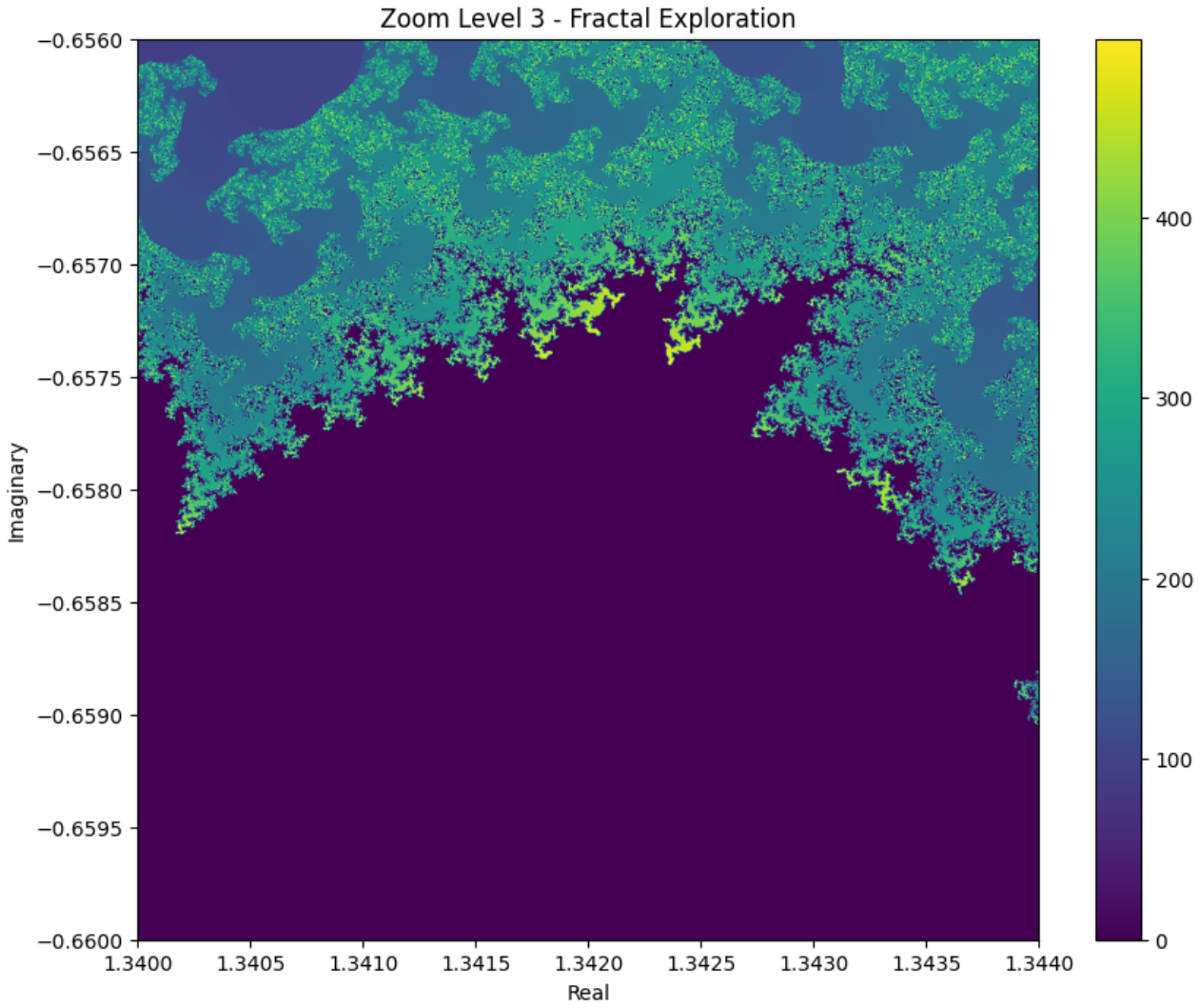


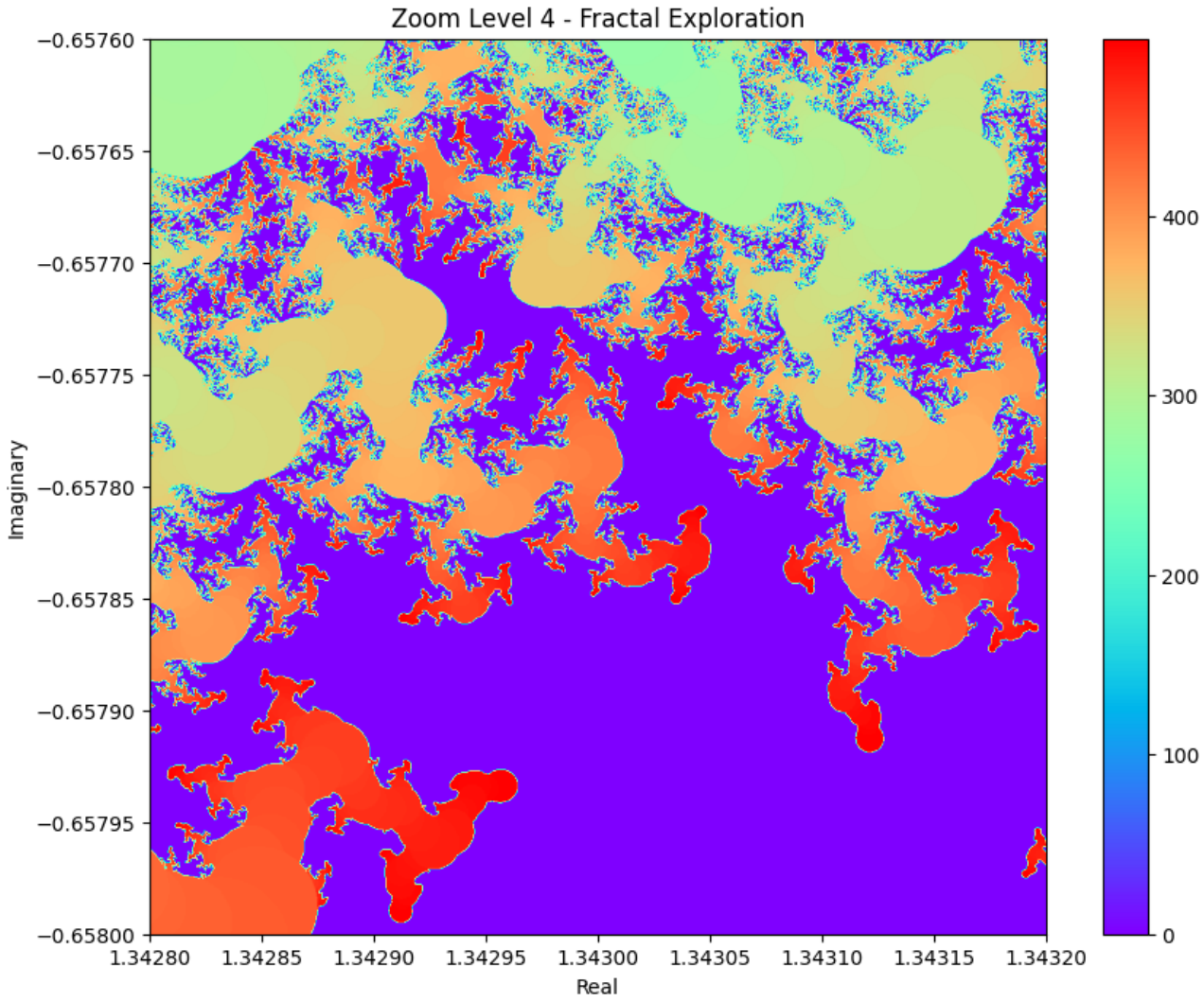

Zoom Level 1 - Fractal Exploration
−0.40
−0.45
−0.50
−0.55
−0.60
−0.65
−0.70
−0.75
−0.80
Imaginary
1.20
1.25
1.30
1.35
1.40
1.45
1.50
1.55
1.60
Real
400
300
200
100
0


Activity Map of Fractal
Imaginary
Real
Activity Measure (Std Dev)
−0.40
−0.45
−0.50
−0.55
−0.60
−0.65
−0.70
−0.75
−0.80
1.20
1.25
1.30
1.35
1.40
1.45
1.50
1.55
1.60
60
50
40
30
20
10
0


Zoom Level 2 - Fractal Exploration
−0.640
−0.645
−0.650
−0.655
−0.660
−0.665
−0.670
−0.675
−0.680
Imaginary
1.320
1.325
1.330
1.335
1.340
1.345
1.350
1.355
1.360
Real
400
300
200
100
0


Activity Map of Fractal
−0.640
−0.645
−0.650
−0.655
−0.660
−0.665
−0.670
−0.675
−0.680
Imaginary
1.32
1.33
1.34
1.35
1.36
Real
120
100
80
60
40
20
0
Activity Measure (Std Dev)


Zoom Level 3 - Fractal Exploration
−0.6560
−0.6565
−0.6570
−0.6575
−0.6580
−0.6585
−0.6590
−0.6595
−0.6600
Imaginary
1.3400
1.3405
1.3410
1.3415
1.3420
1.3425
1.3430
1.3435
1.3440
Real
400
300
200
100
0

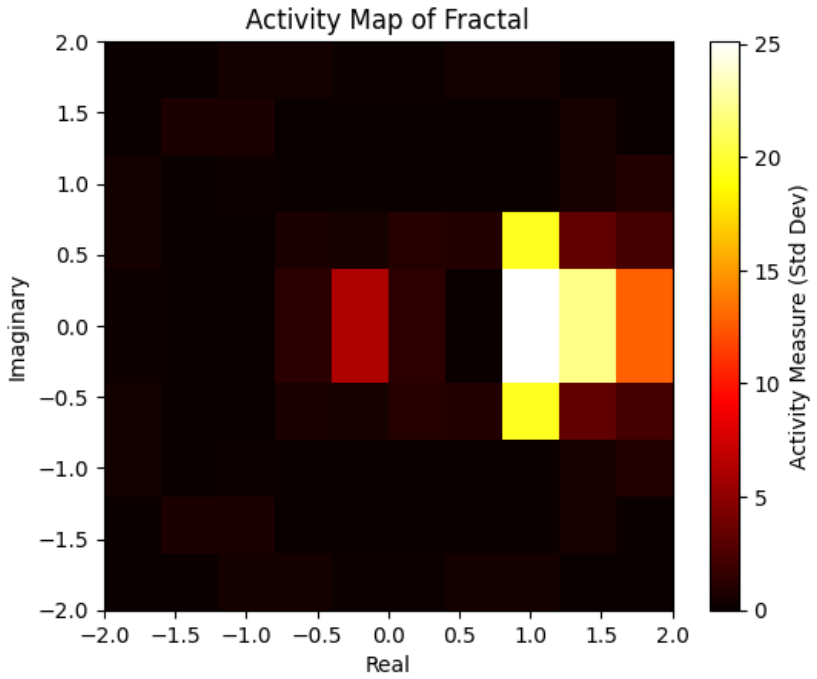


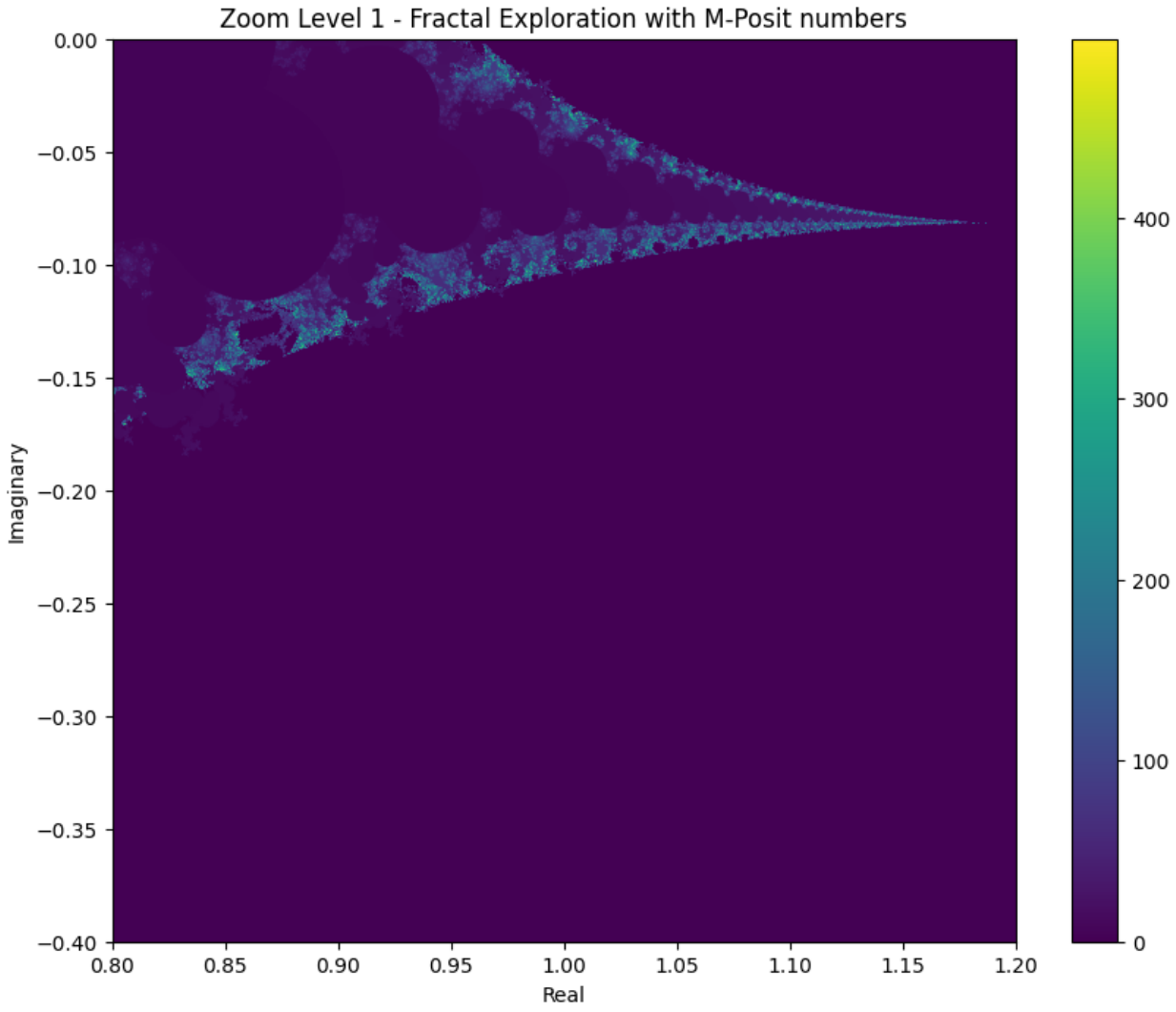

```python
import numpy as np
import matplotlib.pyplot as plt
from numba import jit

# Define the custom fractal function
@jit(nopython=True)
def compute_custom_fractal(real, imag, width, height, max_iter):
    fractal_set = np.zeros((height, width), dtype=np.int32)
    for i in range(height):
        for j in range(width):
            c = complex(real[j], imag[i])
            z = complex(0, 0)
            for k in range(max_iter):
                z = np.sin(z) - c * z + c  # Custom transformation
                if (z.real ** 2 + z.imag ** 2) >= 4:
                    fractal_set[i, j] = k
                    break
    return fractal_set

def measure_activity(fractal_set):
```

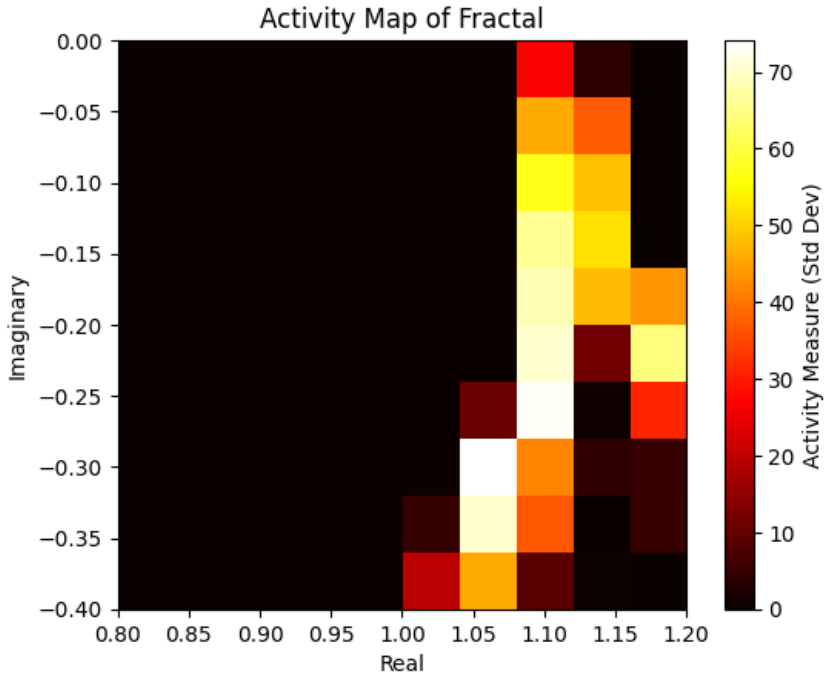
```
    """ Measure activity by calculating the standard deviation of the
        iteration counts. """
    return np.std(fractal_set)

def scan_fractal(x_min, x_max, y_min, y_max, resolution, subregion_size,
    max_iter):
    width = height = resolution
    real = np.linspace(x_min, x_max, width)
    imag = np.linspace(y_min, y_max, height)

    fractal_set = compute_custom_fractal(real, imag, width, height,
        max_iter)

    num_subregions = resolution // subregion_size
    activity_map = np.zeros((num_subregions, num_subregions))

    for i in range(num_subregions):
        for j in range(num_subregions):
            subregion = fractal_set[i*subregion_size:(i+1)*subregion_size,
                j*subregion_size:(j+1)*subregion_size]
            activity_map[i, j] = measure_activity(subregion)

    return activity_map, fractal_set

def plot_activity_map(activity_map, x_min, x_max, y_min, y_max):
    plt.imshow(activity_map, extent=[x_min, x_max, y_min, y_max], origin='
        lower', cmap='hot')
    plt.colorbar(label='Activity Measure (Std Dev)')
    plt.title('Activity Map of Fractal')
    plt.xlabel('Real')
    plt.ylabel('Imaginary')
    plt.show()

def zoom_into_active_region(x_min, x_max, y_min, y_max, resolution,
    subregion_size, max_iter, num_zoom_levels):
    for level in range(num_zoom_levels):
        activity_map, fractal_set = scan_fractal(x_min, x_max, y_min,
            y_max, resolution, subregion_size, max_iter)
        plot_activity_map(activity_map, x_min, x_max, y_min, y_max)

        i, j = np.unravel_index(np.argmax(activity_map), activity_map.
            shape)
        subregion_width = (x_max - x_min) / (resolution // subregion_size)
        subregion_height = (y_max - y_min) / (resolution // subregion_size
            )

        x_min = x_min + j * subregion_width
        x_max = x_min + subregion_width
        y_min = y_min + i * subregion_height
        y_max = y_min + subregion_height

        title = f'Zoom Level {level + 1} - Fractal Exploration'
        explore_fractal(resolution, x_min, x_max, y_min, y_max, max_iter,
            title)
```

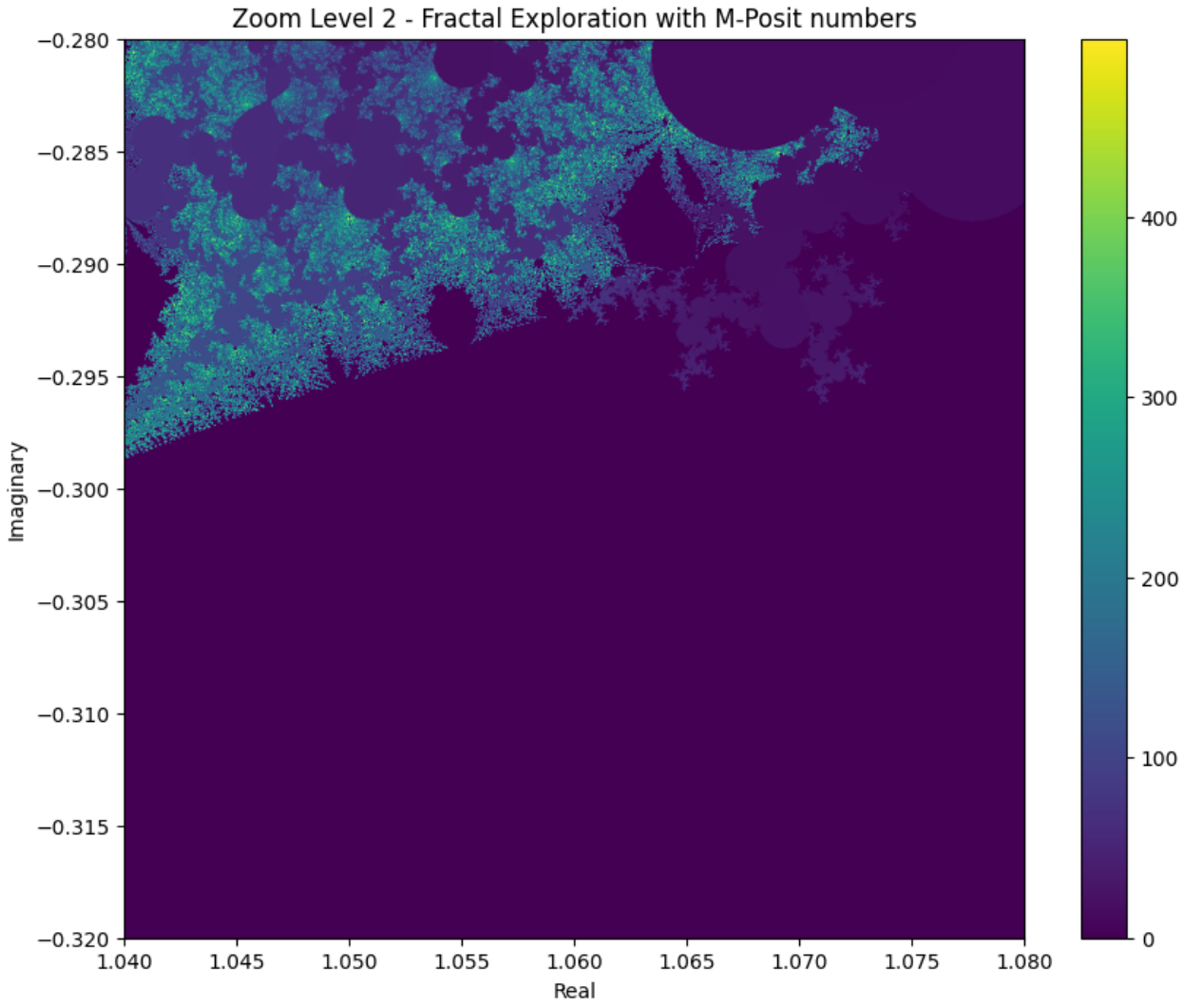
```
65
66  def explore_fractal(n, x_min, x_max, y_min, y_max, max_iter, title):
67      real = np.linspace(x_min, x_max, n)
68      imag = np.linspace(y_min, y_max, n)
69
70      fractal_set = compute_custom_fractal(real, imag, n, n, max_iter)
71
72      plt.figure(figsize=(10, 8))
73      plt.imshow(fractal_set.T, extent=[x_min, x_max, y_min, y_max], cmap='
            viridis', origin='lower')
74      plt.colorbar()
75      plt.title(title)
76      plt.xlabel('Real')
77      plt.ylabel('Imaginary')
78      plt.show()
79
80  # Initial parameters for the broad view
81  x_min, x_max = -2, 2
82  y_min, y_max = -2, 2
83  resolution = 1000
84  subregion_size = 100
85  max_iter = 500
86  num_zoom_levels = 3  # You can adjust the number of zoom levels
87
88  zoom_into_active_region(x_min, x_max, y_min, y_max, resolution,
        subregion_size, max_iter, num_zoom_levels)
```

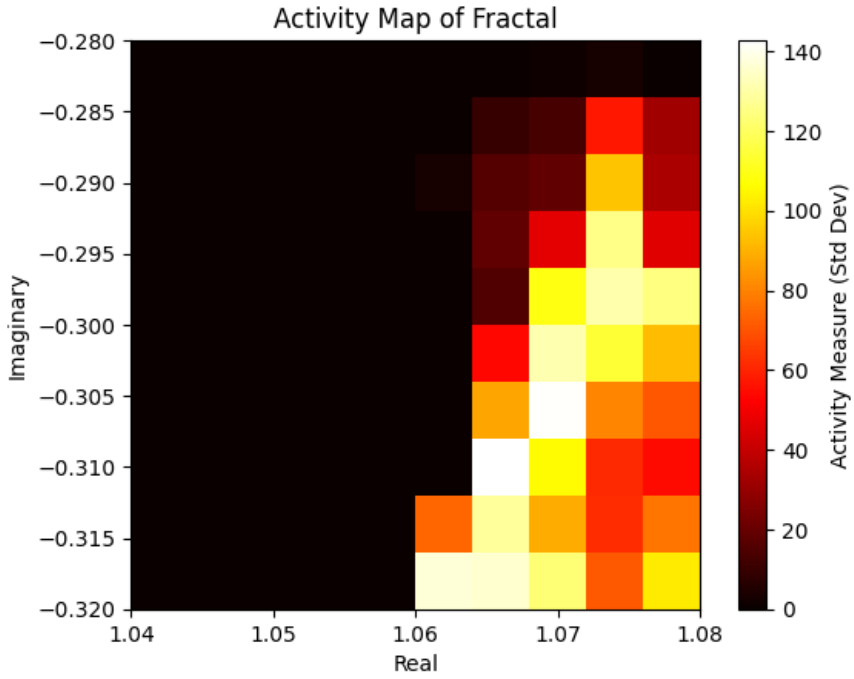
4 Fractal Modification A

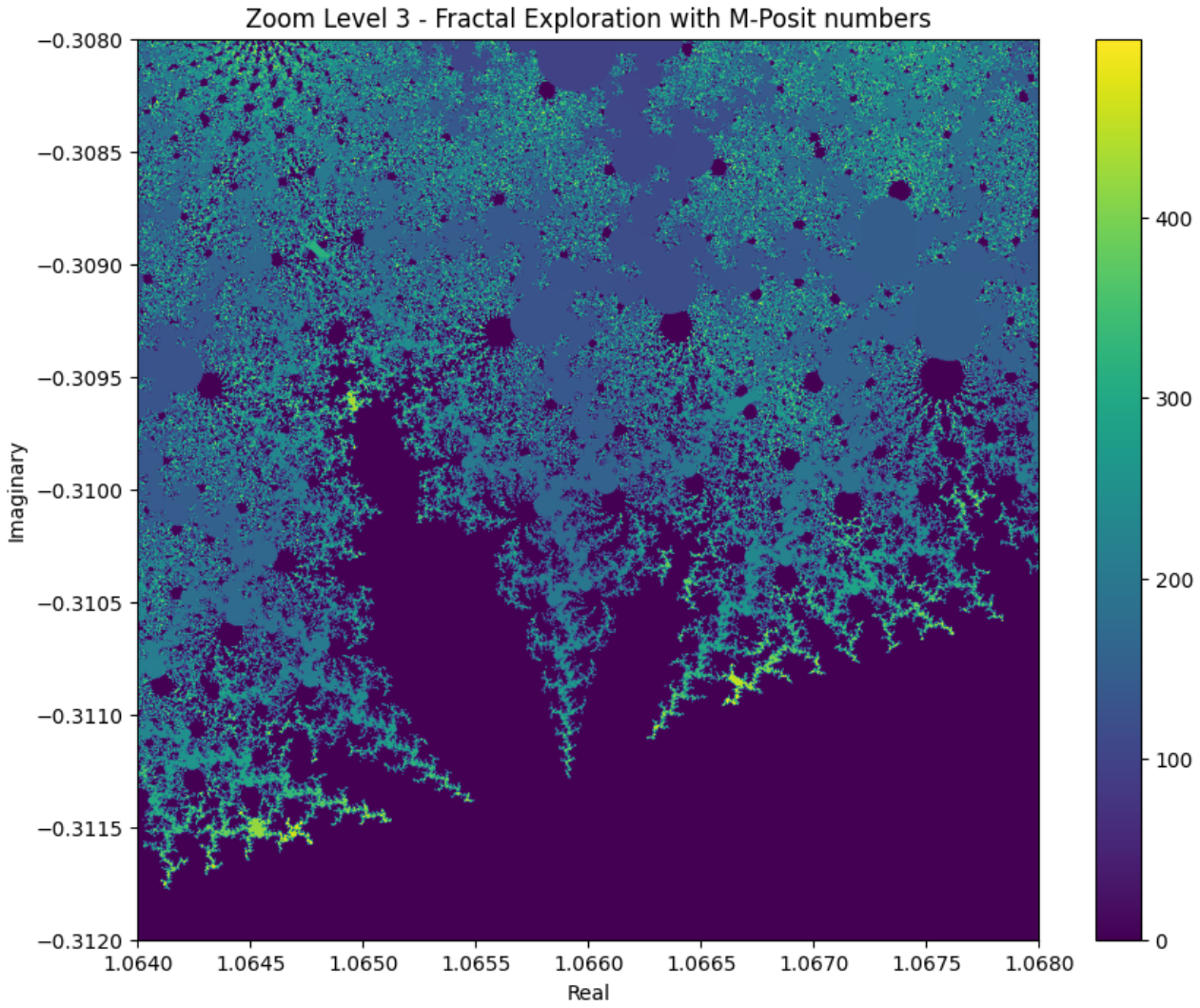



Zoom Level 1 - Fractal Exploration with M-Posit numbers
0.00
−0.05
−0.10
−0.15
−0.20
−0.25
−0.30
−0.35
−0.40
Imaginary
0.80
0.85
0.90
0.95
1.00
1.05
1.10
1.15
1.20
Real
400
300
200
100
0


Activity Map of Fractal
Imaginary
Real
Activity Measure (Std Dev)
0.00
−0.05
−0.10
−0.15
−0.20
−0.25
−0.30
−0.35
−0.40
0.80
0.85
0.90
0.95
1.00
1.05
1.10
1.15
1.20
70
60
50
40
30
20
10
0


Zoom Level 2 - Fractal Exploration with M-Posit numbers
Imaginary
Real
−0.280
−0.285
−0.290
−0.295
−0.300
−0.305
−0.310
−0.315
−0.320
1.040
1.045
1.050
1.055
1.060
1.065
1.070
1.075
1.080
0
100
200
300
400


Activity Map of Fractal
−0.280
−0.285
−0.290
−0.295
−0.300
−0.305
−0.310
−0.315
−0.320
Imaginary
1.04
1.05
1.06
1.07
1.08
Real
140
120
100
80
60
40
20
0
Activity Measure (Std Dev)

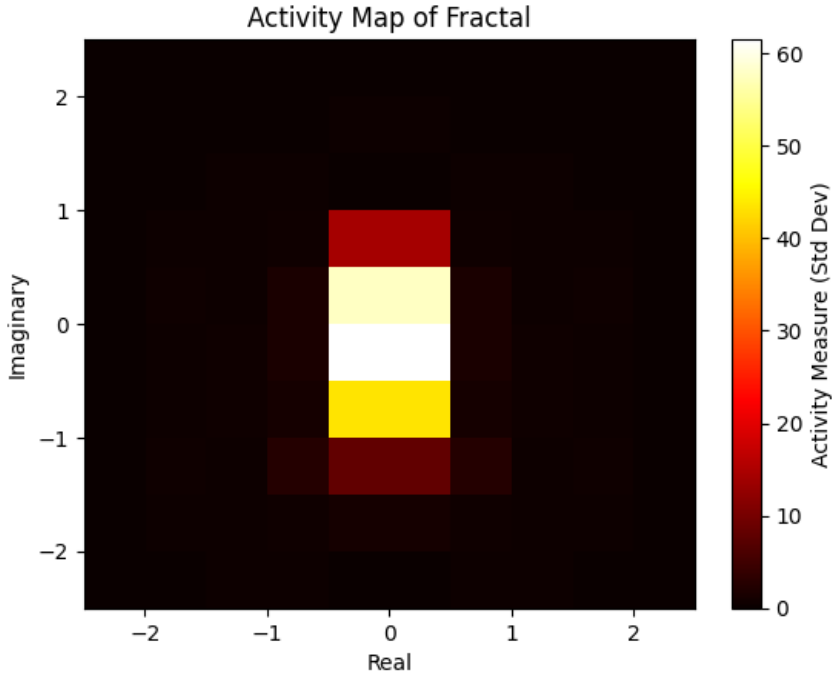


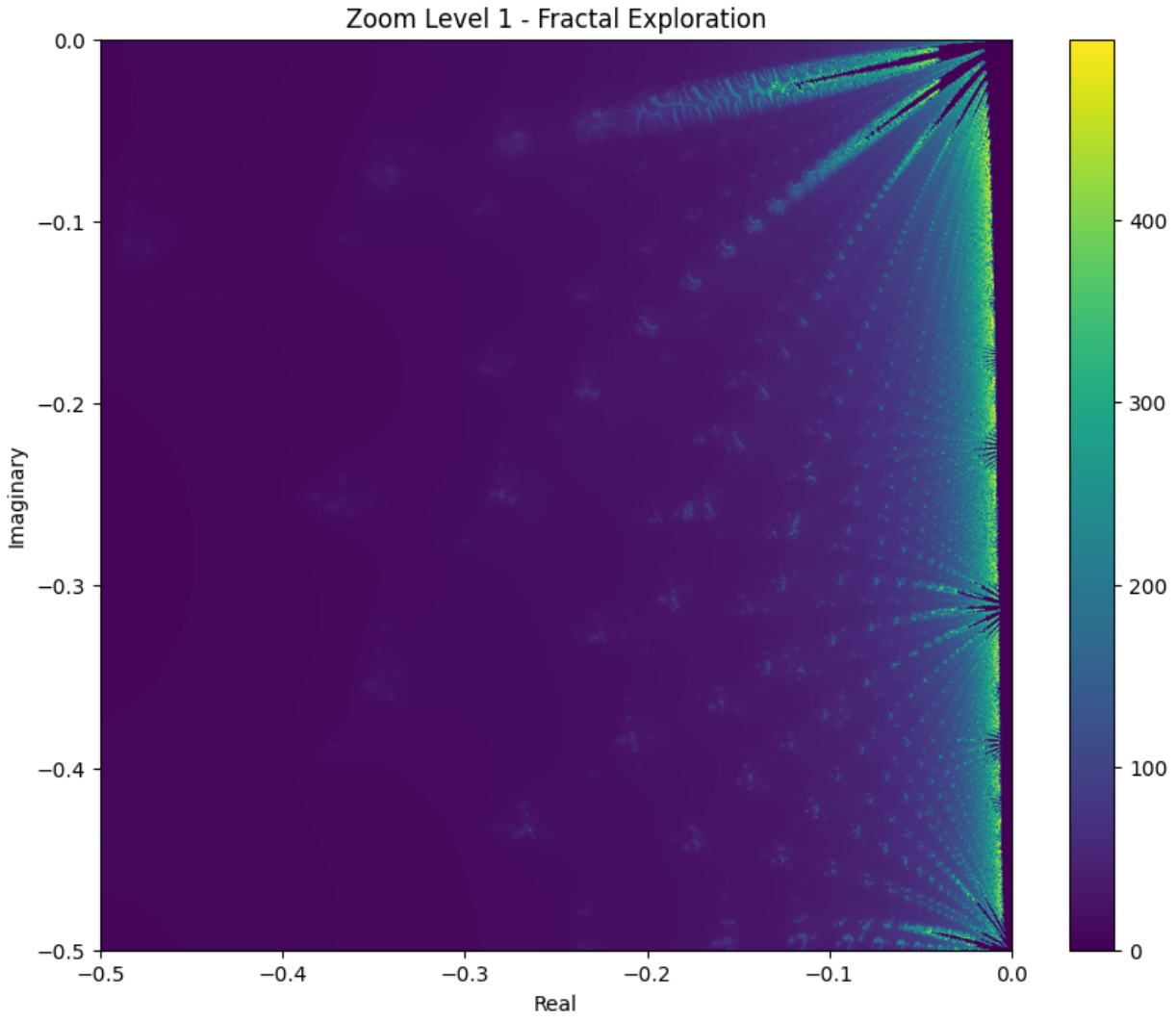

```python
import numpy as np
import matplotlib.pyplot as plt
from numba import jit

# Define a custom fractal function with M-posit numbers
@jit(nopython=True)
def compute_custom_fractal(real, imag, width, height, max_iter):
    fractal_set = np.zeros((height, width), dtype=np.int32)
    nu_E = 0.5  # Use a small epsilon to represent  E
    for i in range(height):
        for j in range(width):
            c = complex(real[j], imag[i])
            z = complex(0, 0)
            for k in range(max_iter):
                # Introduce quantum and differential perturbations
                z = np.sin(z) - c * z + c + nu_E * (np.sin(np.pi * z) / np
                    .pi)  # Custom transformation with M-posit number
                    concept
                if (z.real ** 2 + z.imag ** 2) >= 4:
                    fractal_set[i, j] = k
```

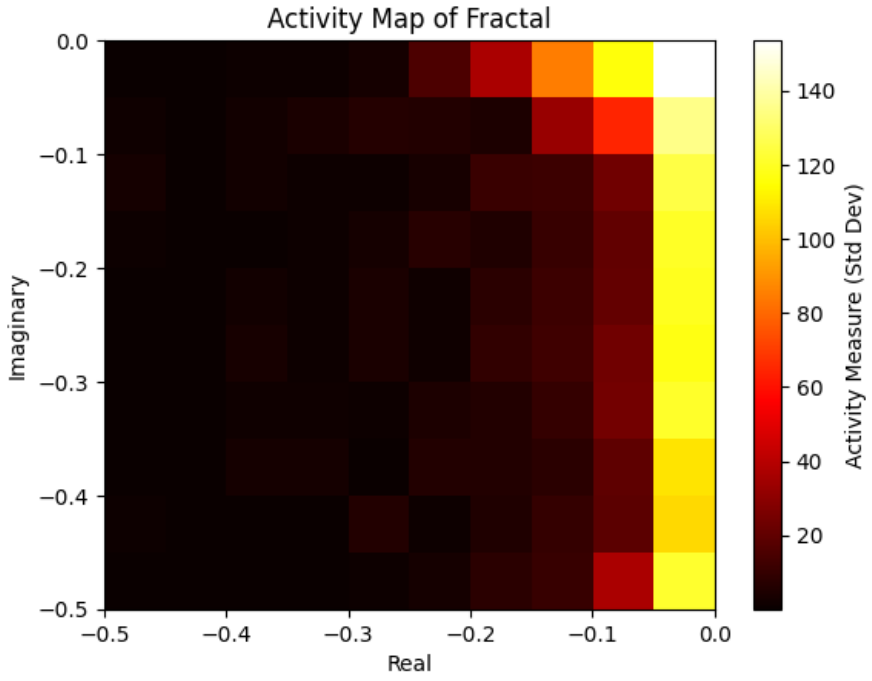
```
                    break
    return fractal_set

def measure_activity(fractal_set):
    """ Measure activity by calculating the standard deviation of the
        iteration counts. """
    return np.std(fractal_set)

def scan_fractal(x_min, x_max, y_min, y_max, resolution, subregion_size,
    max_iter):
    width = height = resolution
    real = np.linspace(x_min, x_max, width)
    imag = np.linspace(y_min, y_max, height)

    fractal_set = compute_custom_fractal(real, imag, width, height,
        max_iter)

    num_subregions = resolution // subregion_size
    activity_map = np.zeros((num_subregions, num_subregions))

    for i in range(num_subregions):
        for j in range(num_subregions):
            subregion = fractal_set[i*subregion_size:(i+1)*subregion_size,
                j*subregion_size:(j+1)*subregion_size]
            activity_map[i, j] = measure_activity(subregion)

    return activity_map, fractal_set

def plot_activity_map(activity_map, x_min, x_max, y_min, y_max):
    plt.imshow(activity_map, extent=[x_min, x_max, y_min, y_max], origin='
        lower', cmap='hot')
    plt.colorbar(label='Activity␣Measure␣(Std␣Dev)')
    plt.title('Activity␣Map␣of␣Fractal')
    plt.xlabel('Real')
    plt.ylabel('Imaginary')
    plt.show()

def zoom_into_active_region(x_min, x_max, y_min, y_max, resolution,
    subregion_size, max_iter, num_zoom_levels):
    for level in range(num_zoom_levels):
        activity_map, fractal_set = scan_fractal(x_min, x_max, y_min,
            y_max, resolution, subregion_size, max_iter)
        plot_activity_map(activity_map, x_min, x_max, y_min, y_max)

        i, j = np.unravel_index(np.argmax(activity_map), activity_map.
            shape)
        subregion_width = (x_max - x_min) / (resolution // subregion_size)
        subregion_height = (y_max - y_min) / (resolution // subregion_size
            )

        x_min, x_max = x_min + j * subregion_width, x_min + (j + 1) *
            subregion_width
        y_min, y_max = y_min + i * subregion_height, y_min + (i + 1) *
            subregion_height
```

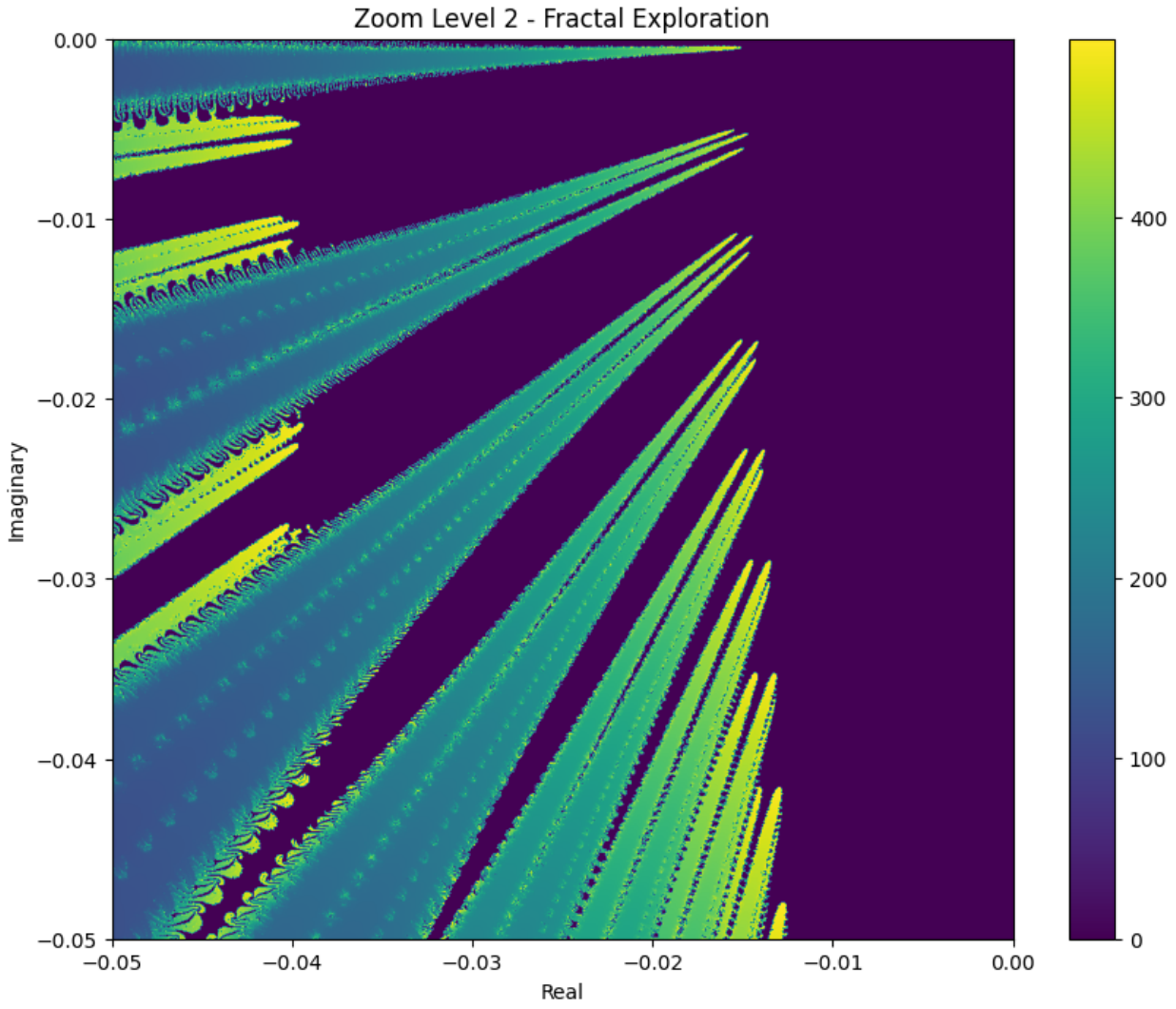
```
62
63          title = f'Zoom␣Level␣{level␣+␣1}␣-␣Fractal␣Exploration␣with␣M-
                Posit␣numbers'
64          explore_fractal(resolution, x_min, x_max, y_min, y_max, max_iter,
                title)
65
66  def explore_fractal(n, x_min, x_max, y_min, y_max, max_iter, title):
67      real = np.linspace(x_min, x_max, n)
68      imag = np.linspace(y_min, y_max, n)
69
70      fractal_set = compute_custom_fractal(real, imag, n, n, max_iter)
71
72      plt.figure(figsize=(10, 8))  # Larger figure to show more details
73      plt.imshow(fractal_set.T, extent=[x_min, x_max, y_min, y_max], cmap='
            viridis', origin='lower')
74      plt.colorbar()
75      plt.title(title)
76      plt.xlabel('Real')
77      plt.ylabel('Imaginary')
78      plt.show()
79
80  # Initial parameters for the broad view
81  x_min, x_max = -2, 2
82  y_min, y_max = -2, 2
83  resolution = 1000
84  subregion_size = 100
85  max_iter = 500
86  num_zoom_levels = 10  # Set the number of zoom levels to 10
87
88  zoom_into_active_region(x_min, x_max, y_min, y_max, resolution,
        subregion_size, max_iter, num_zoom_levels)
```

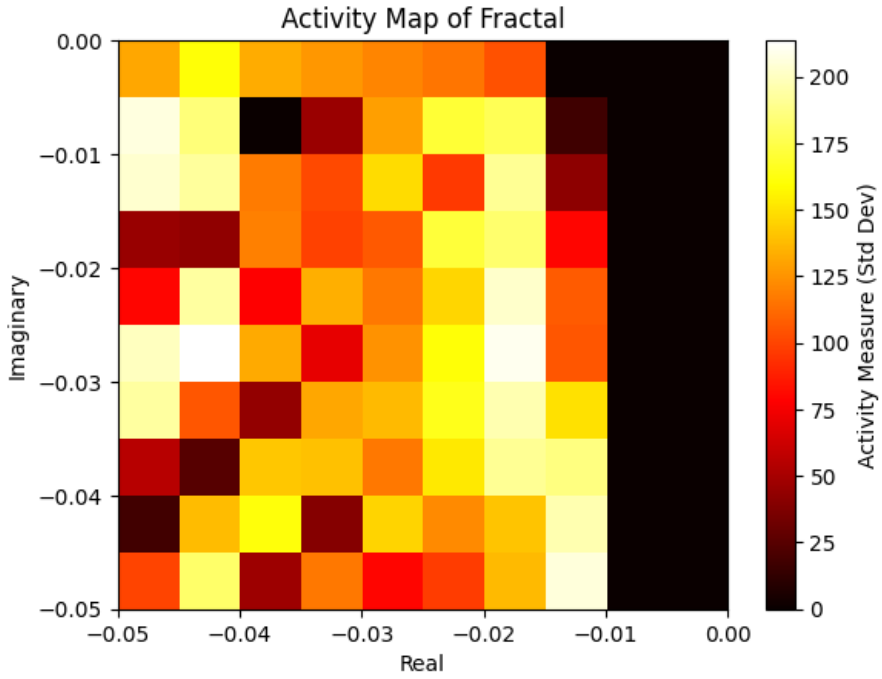
5 Fractal Modification B

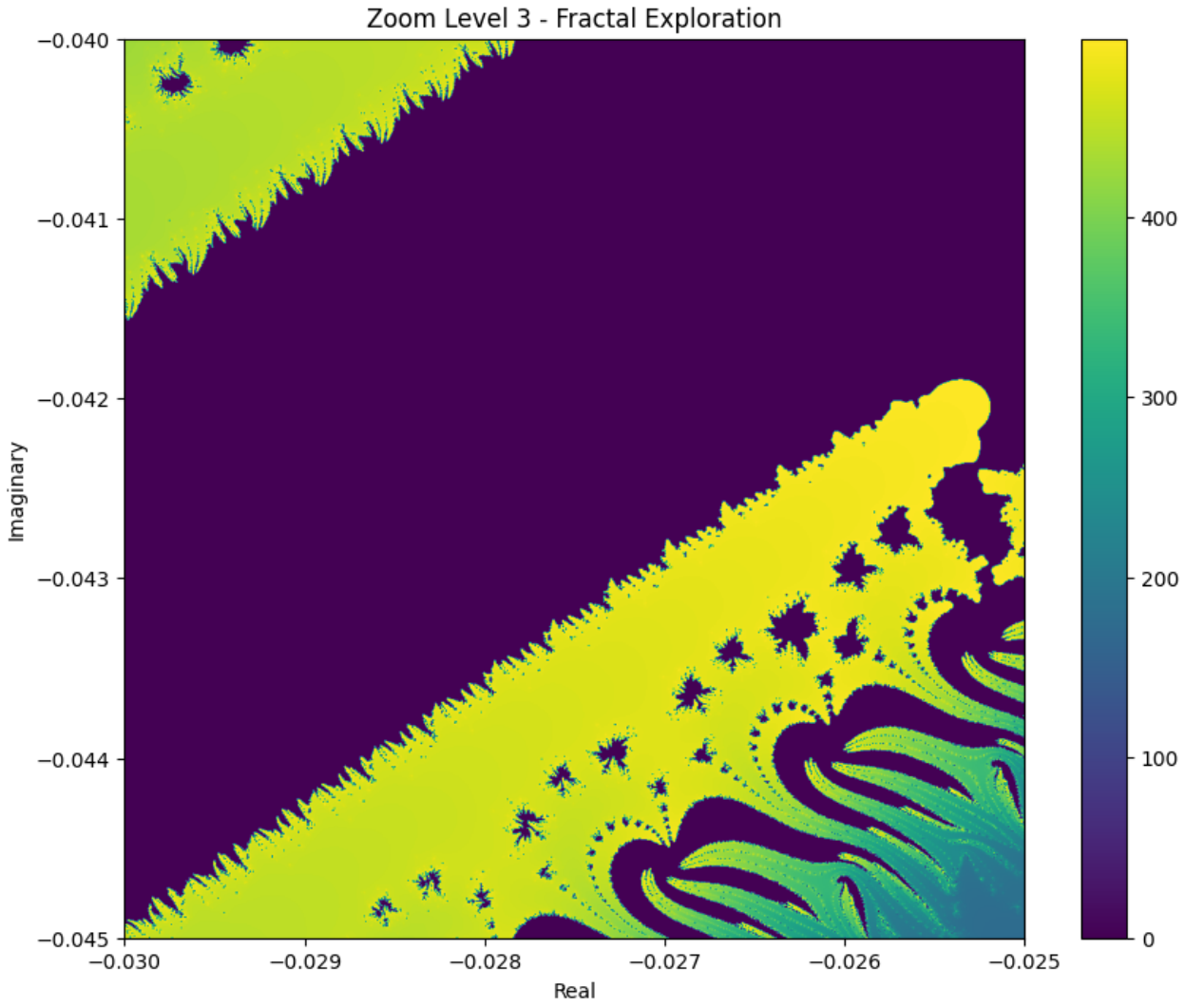



Zoom Level 1 - Fractal Exploration
0.0
−0.1
−0.2
−0.3
−0.4
−0.5
Imaginary
−0.5
−0.4
−0.3
−0.2
−0.1
0.0
Real
400
300
200
100
0


Activity Map of Fractal
Imaginary
Real
Activity Measure (Std Dev)
0.0
−0.1
−0.2
−0.3
−0.4
−0.5
−0.5
−0.4
−0.3
−0.2
−0.1
0.0
140
120
100
80
60
40
20


Zoom Level 2 - Fractal Exploration
0.00
−0.01
−0.02
−0.03
−0.04
−0.05
Imaginary
−0.05
−0.04
−0.03
−0.02
−0.01
0.00
Real
400
300
200
100
0


Activity Map of Fractal
Imaginary
Real
Activity Measure (Std Dev)
0.00
−0.01
−0.02
−0.03
−0.04
−0.05
−0.05
−0.04
−0.03
−0.02
−0.01
0.00
200
175
150
125
100
75
50
25
0

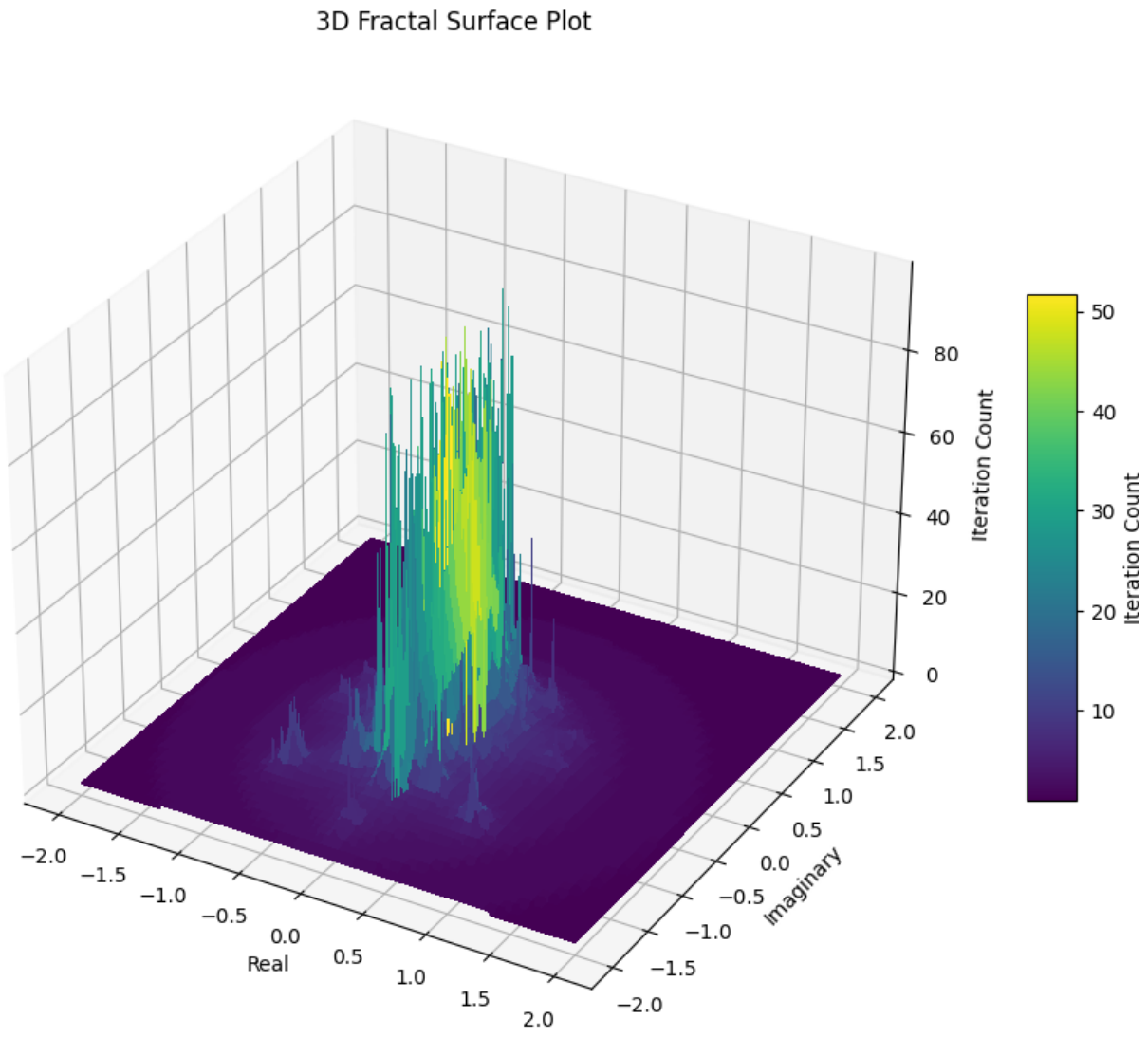



```python
import numpy as np
import matplotlib.pyplot as plt
from numba import jit
import warnings

# Suppress warnings from invalid operations (e.g., overflow)
warnings.filterwarnings("ignore", category=RuntimeWarning)

# Define the custom fractal function with the new iterative function
@jit(nopython=True)
def compute_custom_fractal(real, imag, width, height, max_iter):
    fractal_set = np.zeros((height, width), dtype=np.int32)
    for i in range(height):
        for j in range(width):
            c = complex(real[j], imag[i])
            z = complex(0, 0)
            z_prev = complex(0, 0)  # Initialize z_{n-1}
            for k in range(max_iter):
                z_temp = z  # Store current z to update z_prev later
                z = z**2 - z_prev + c  # New iterative function
```

```
                z_prev = z_temp  # Update z_prev for the next iteration
                if abs(z) >= 4:
                    fractal_set[i, j] = k
                    break
    return fractal_set

def measure_activity(fractal_set):
    """ Measure activity by calculating the standard deviation of the
        iteration counts. """
    return np.std(fractal_set)

def scan_fractal(x_min, x_max, y_min, y_max, resolution, subregion_size,
    max_iter):
    width = height = resolution
    real = np.linspace(x_min, x_max, width)
    imag = np.linspace(y_min, y_max, height)

    fractal_set = compute_custom_fractal(real, imag, width, height,
        max_iter)

    num_subregions = resolution // subregion_size
    activity_map = np.zeros((num_subregions, num_subregions))

    for i in range(num_subregions):
        for j in range(num_subregions):
            subregion = fractal_set[
                i*subregion_size:(i+1)*subregion_size,
                j*subregion_size:(j+1)*subregion_size
            ]
            activity_map[i, j] = measure_activity(subregion)

    return activity_map, fractal_set

def plot_activity_map(activity_map, x_min, x_max, y_min, y_max):
    extent = [x_min, x_max, y_min, y_max]
    plt.imshow(activity_map.T, extent=extent, origin='lower', cmap='hot',
        aspect='auto')
    plt.colorbar(label='Activity␣Measure␣(Std␣Dev)')
    plt.title('Activity␣Map␣of␣Fractal')
    plt.xlabel('Real')
    plt.ylabel('Imaginary')
    plt.show()

def explore_fractal(n, x_min, x_max, y_min, y_max, max_iter, title):
    real = np.linspace(x_min, x_max, n)
    imag = np.linspace(y_min, y_max, n)

    fractal_set = compute_custom_fractal(real, imag, n, n, max_iter)

    plt.figure(figsize=(10, 8))
    plt.imshow(fractal_set.T, extent=[x_min, x_max, y_min, y_max], cmap='
        viridis', origin='lower')
    plt.colorbar()
    plt.title(title)
```

```
    plt.xlabel('Real')
    plt.ylabel('Imaginary')
    plt.show()

def zoom_into_active_region(x_min, x_max, y_min, y_max, resolution,
    subregion_size, max_iter, num_zoom_levels):
    for level in range(num_zoom_levels):
        print(f"Zoom Level {level+1} - Scanning Region")
        activity_map, fractal_set = scan_fractal(x_min, x_max, y_min,
            y_max, resolution, subregion_size, max_iter)
        plot_activity_map(activity_map, x_min, x_max, y_min, y_max)

        # Find the subregion with the highest activity
        i, j = np.unravel_index(np.argmax(activity_map), activity_map.
            shape)
        subregion_width = (x_max - x_min) / (resolution // subregion_size)
        subregion_height = (y_max - y_min) / (resolution // subregion_size
            )

        # Update region boundaries to zoom in on the active subregion
        x_min_new = x_min + j * subregion_width
        x_max_new = x_min_new + subregion_width
        y_min_new = y_min + i * subregion_height
        y_max_new = y_min_new + subregion_height

        # Update the zoomed region for the next iteration
        x_min, x_max = x_min_new, x_max_new
        y_min, y_max = y_min_new, y_max_new

        title = f'Zoom Level {level + 1} - Fractal Exploration'
        explore_fractal(resolution, x_min, x_max, y_min, y_max, max_iter,
            title)

# Parameters for the initial broad view
x_min, x_max = -2.5, 2.5
y_min, y_max = -2.5, 2.5
resolution = 1000
subregion_size = 100
max_iter = 500
num_zoom_levels = 9  # Adjust the number of zoom levels as desired

# Start the fractal exploration
zoom_into_active_region(x_min, x_max, y_min, y_max, resolution,
    subregion_size, max_iter, num_zoom_levels)
```

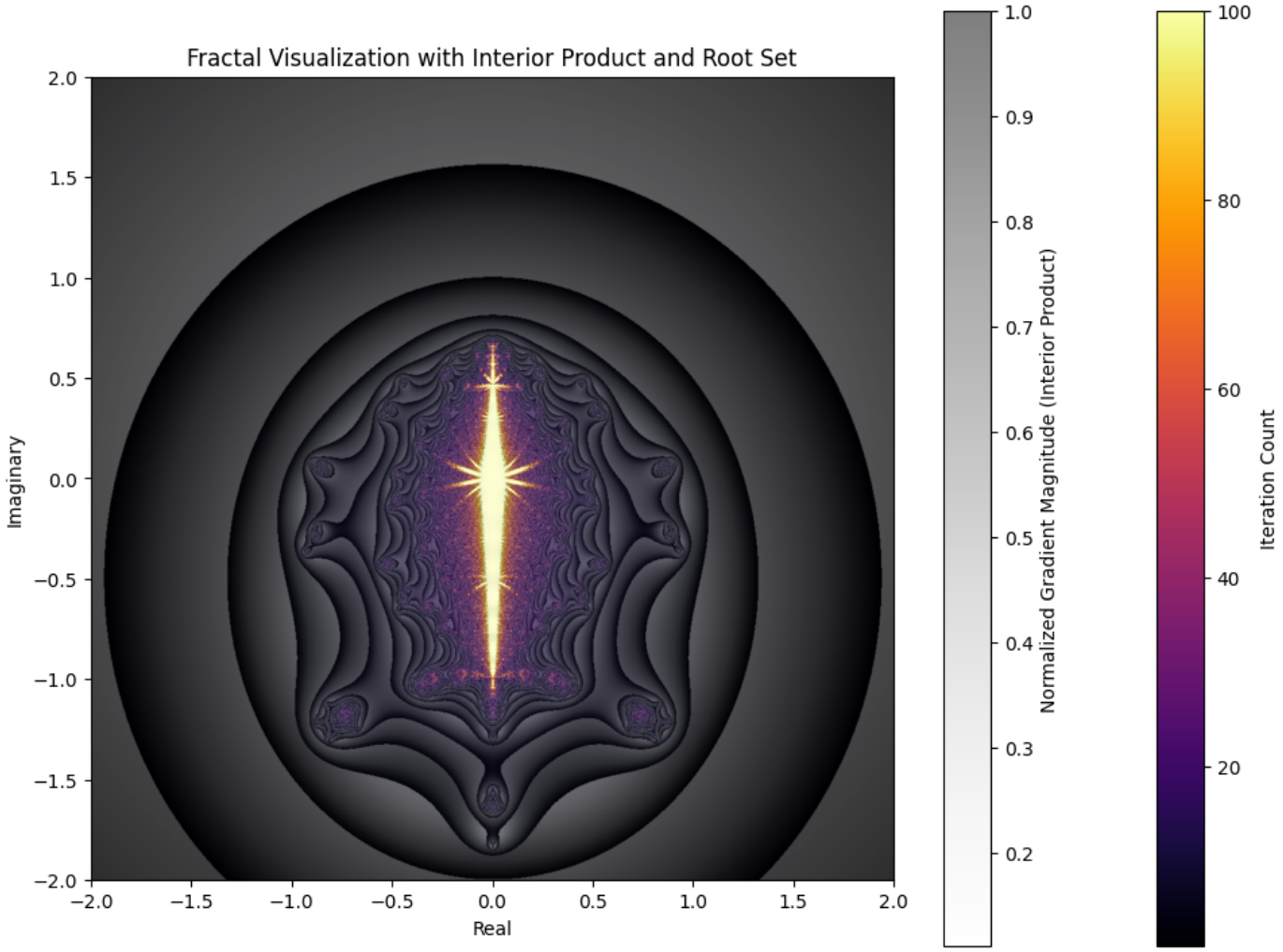
6 3D Implementation




```python
import numpy as np
import matplotlib.pyplot as plt
from numba import jit
from mpl_toolkits.mplot3d import Axes3D  # Import 3D plotting toolkit
import warnings

# Suppress warnings from invalid operations (e.g., overflow)
warnings.filterwarnings("ignore", category=RuntimeWarning)

# Define the custom fractal function with the iterative function
@jit(nopython=True)
def compute_custom_fractal(real, imag, width, height, max_iter):
    fractal_set = np.zeros((height, width), dtype=np.int32)
    for i in range(height):
        for j in range(width):
            c = complex(real[j], imag[i])
```

```
            z = complex(0, 0)
            z_prev = complex(0, 0)  # Initialize z_{n-1}
            for k in range(max_iter):
                z_temp = z  # Store current z to update z_prev later
                z = z**2 - z_prev + c  # Iterative function
                z_prev = z_temp  # Update z_prev for the next iteration
                if abs(z) >= 4:
                    fractal_set[i, j] = k
                    break
    return fractal_set

def plot_fractal_3d(x_min, x_max, y_min, y_max, resolution, max_iter):
    real = np.linspace(x_min, x_max, resolution)
    imag = np.linspace(y_min, y_max, resolution)

    # Compute the fractal set
    fractal_set = compute_custom_fractal(real, imag, resolution,
        resolution, max_iter)

    # Create meshgrid for plotting
    X, Y = np.meshgrid(real, imag)
    Z = fractal_set.T  # Transpose to align axes correctly

    # Create a 3D plot
    fig = plt.figure(figsize=(12, 9))
    ax = fig.add_subplot(111, projection='3d')

    # Plot the surface
    surf = ax.plot_surface(X, Y, Z, cmap='viridis', linewidth=0,
        antialiased=False)

    # Customize the axes and labels
    ax.set_xlabel('Real')
    ax.set_ylabel('Imaginary')
    ax.set_zlabel('Iteration␣Count')
    ax.set_title('3D␣Fractal␣Surface␣Plot')

    # Add a color bar which maps values to colors
    fig.colorbar(surf, shrink=0.5, aspect=10, label='Iteration␣Count')

    plt.show()

# Parameters for the fractal
x_min, x_max = -2, 2
y_min, y_max = -2, 2
resolution = 500  # Adjust for desired detail and performance
max_iter = 100

# Generate and plot the 3D fractal
plot_fractal_3d(x_min, x_max, y_min, y_max, resolution, max_iter)
```

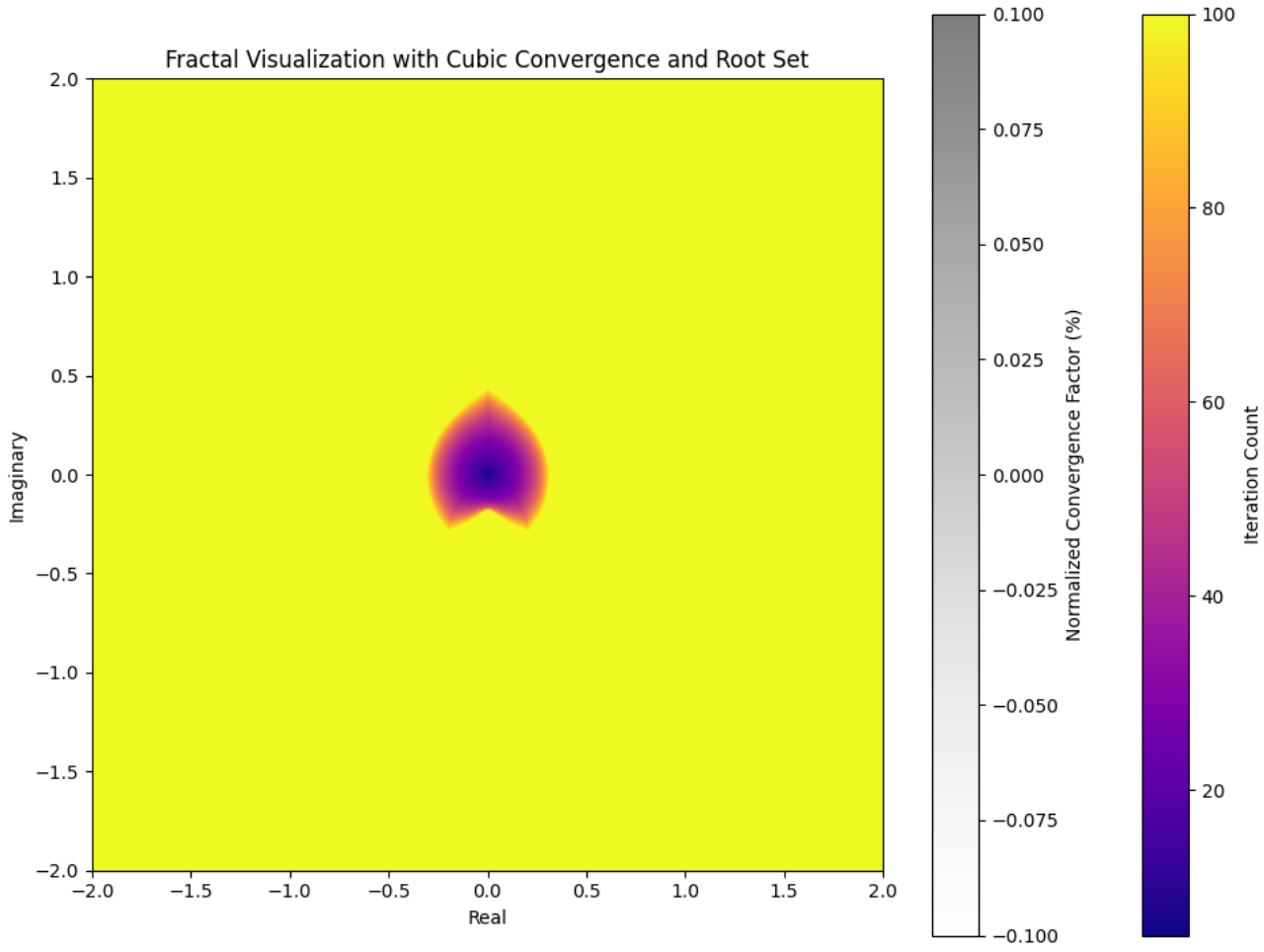
7 Applying "The exterior product of a morphism is an interior product of its root set," to Fractals




```python
import numpy as np
import matplotlib.pyplot as plt
from numba import njit
import warnings

# Suppress warnings from invalid operations
warnings.filterwarnings("ignore", category=RuntimeWarning)

# Define the custom fractal function with the iterative function
@njit
def compute_fractal_with_interior_product(x_min, x_max, y_min, y_max,
    width, height, max_iter):
    # Create arrays for the real and imaginary parts
    real = np.linspace(x_min, x_max, width)
    imag = np.linspace(y_min, y_max, height)

    # Initialize arrays for the fractal set, divergence (interior product
```

```
        analogy), and gradient magnitude
    fractal_set = np.zeros((height, width), dtype=np.int32)
    divergence = np.zeros((height, width), dtype=np.float64)

    for i in range(height):
        for j in range(width):
            c = complex(real[j], imag[i])
            z = complex(0, 0)
            z_prev = complex(0, 0)  # Initialize z_{n-1}
            diverged = False
            for k in range(max_iter):
                # Calculate the derivatives (gradients)
                df_dz_n = 2 * z
                df_dz_n_minus_1 = -1
                grad_magnitude = np.abs(df_dz_n) + np.abs(df_dz_n_minus_1)

                # Update z using the iterative function
                z_temp = z  # Store current z to update z_prev later
                z = z**2 - z_prev + c  # Iterative function
                z_prev = z_temp  # Update z_prev for the next iteration

                # Check for divergence
                if abs(z) >= 4:
                    fractal_set[i, j] = k
                    divergence[i, j] = grad_magnitude
                    diverged = True
                    break

            if not diverged:
                # Point is in the root set (does not diverge)
                fractal_set[i, j] = max_iter
                divergence[i, j] = grad_magnitude

    return fractal_set, divergence

def plot_fractal_with_interior_product(x_min, x_max, y_min, y_max,
    resolution, max_iter):
    # Compute the fractal set and divergence
    fractal_set, divergence = compute_fractal_with_interior_product(
        x_min, x_max, y_min, y_max, resolution, resolution, max_iter
    )

    # Normalize the divergence for visualization
    normalized_divergence = divergence / np.max(divergence)

    # Create the plot
    plt.figure(figsize=(12, 9))
    plt.imshow(
        fractal_set.T,
        extent=(x_min, x_max, y_min, y_max),
        cmap='inferno',
        interpolation='bilinear',
        origin='lower'
    )
```

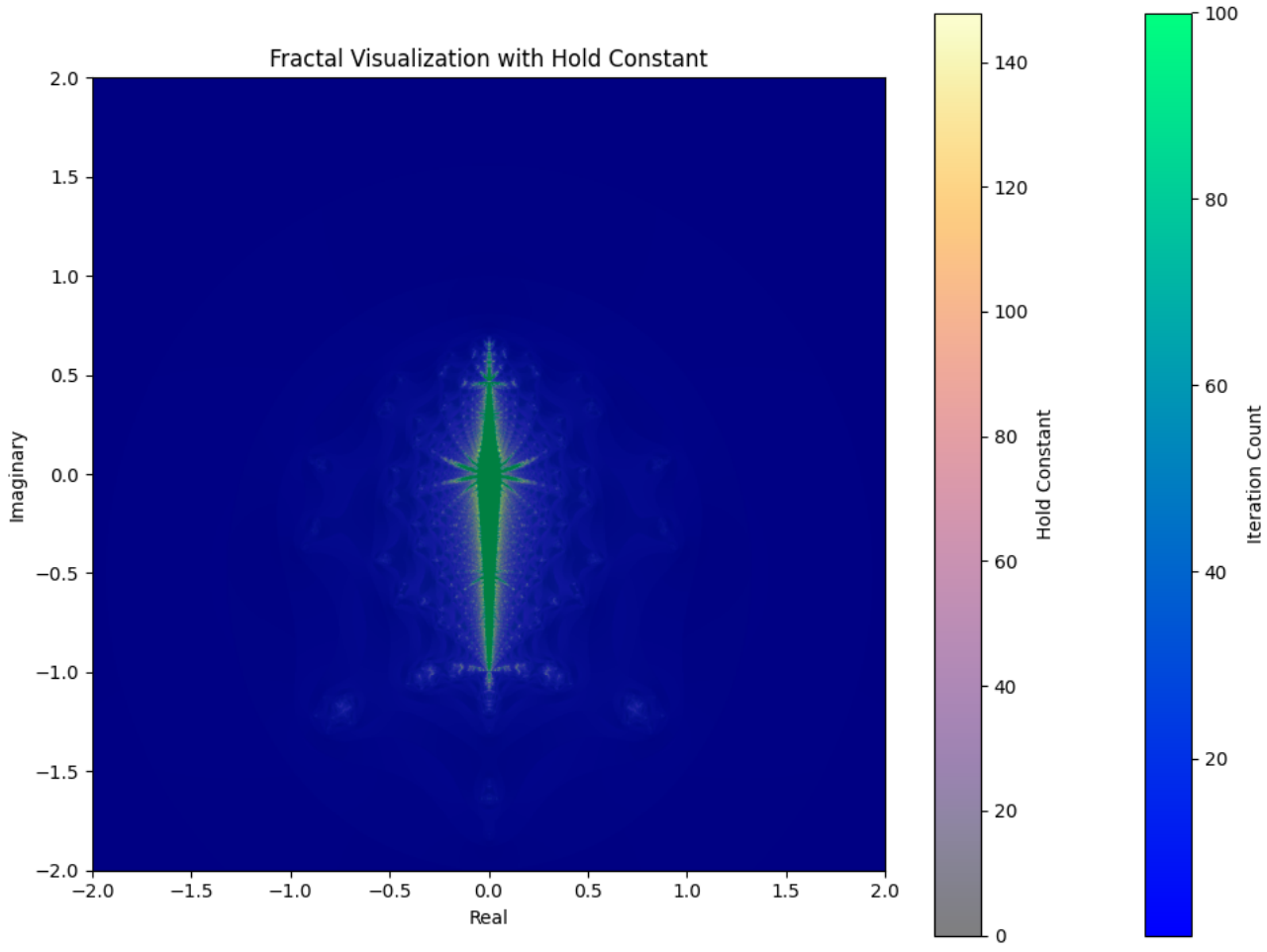
```
    plt.colorbar(label='Iteration Count')
    plt.title('Fractal Visualization with Interior Product and Root Set')
    plt.xlabel('Real')
    plt.ylabel('Imaginary')

    # Overlay the divergence (interior product analogy)
    plt.imshow(
        normalized_divergence.T,
        extent=(x_min, x_max, y_min, y_max),
        cmap='Greys',
        alpha=0.5,
        interpolation='bilinear',
        origin='lower'
    )
    plt.colorbar(label='Normalized Gradient Magnitude (Interior Product)')

    plt.show()

# Parameters for the fractal
x_min, x_max = -2, 2
y_min, y_max = -2, 2
resolution = 1000  # Adjust for desired detail and performance
max_iter = 100

# Generate and plot the fractal with interior product visualization
plot_fractal_with_interior_product(x_min, x_max, y_min, y_max, resolution,
    max_iter)
```




```python
import numpy as np
import matplotlib.pyplot as plt
from numba import njit
import warnings

# Suppress warnings from invalid operations
warnings.filterwarnings("ignore", category=RuntimeWarning)

# Define the custom fractal function with the iterative function
@njit
def compute_fractal_with_cubic_convergence(real_range, imag_range, width, height, max_iter):
    # Initialize arrays for the fractal set and convergence factor (cubic convergence analogy)
    fractal_set = np.zeros((height, width), dtype=np.int32)
    convergence_factor = np.zeros((height, width), dtype=np.float64)

    for i in range(height):
        for j in range(width):
            real = real_range[j]
            imag = imag_range[i]
            c = complex(real, imag)
            z = complex(0, 0)
```

```
            z_prev = complex(0, 0)  # Initialize z_{n-1}
            cubic_converged = False
            for k in range(max_iter):
                # Update z using the iterative function
                z_temp = z  # Store current z to update z_prev later
                z = z**3 - z_prev**2 + c  # Iterative function
                z_prev = z_temp  # Update z_prev for the next iteration

                # Check for cubic convergence (i.e., the ratio of z to
                    z_prev approaches 1)
                if z_prev != 0 and abs(z / z_prev - 1) <= 1e-8:
                    fractal_set[i, j] = k
                    convergence_factor[i, j] = abs(z / z_prev - 1)
                    cubic_converged = True
                    break

            if not cubic_converged:
                # Point is in the root set (does not converge)
                fractal_set[i, j] = max_iter
                convergence_factor[i, j] = abs(z / z_prev - 1)

    return fractal_set, convergence_factor

def plot_fractal_with_cubic_convergence(x_min, x_max, y_min, y_max,
    resolution, max_iter):
    # Create arrays for the real and imaginary parts
    real = np.linspace(x_min, x_max, resolution)
    imag = np.linspace(y_min, y_max, resolution)

    # Compute the fractal set and convergence factor
    fractal_set, convergence_factor =
        compute_fractal_with_cubic_convergence(
        real, imag, resolution, resolution, max_iter
    )

    # Normalize the convergence factor for visualization
    normalized_convergence = convergence_factor / np.max(
        convergence_factor) * 100

    # Create the plot
    plt.figure(figsize=(12, 9))
    plt.imshow(
        fractal_set.T,
        extent=(x_min, x_max, y_min, y_max),
        cmap='plasma',
        interpolation='bilinear',
        origin='lower'
    )
    plt.colorbar(label='Iteration␣Count')
    plt.title('Fractal␣Visualization␣with␣Cubic␣Convergence␣and␣Root␣Set')
    plt.xlabel('Real')
    plt.ylabel('Imaginary')

    # Overlay the convergence factor (cubic convergence analogy)
```

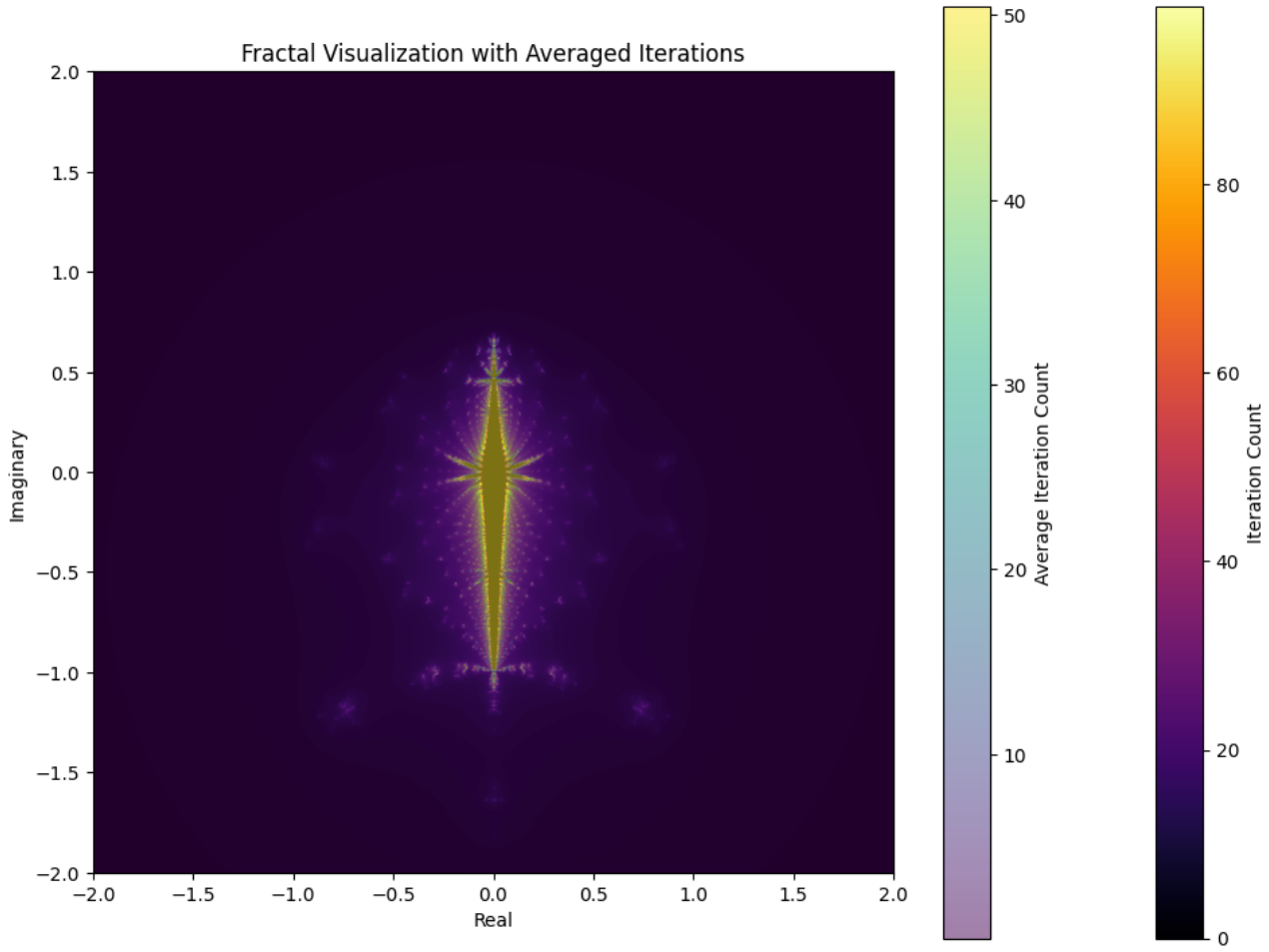
```
    plt.imshow(
        normalized_convergence.T,
        extent=(x_min, x_max, y_min, y_max),
        cmap='Greys',
        alpha=0.5,
        interpolation='bilinear',
        origin='lower'
    )
    plt.colorbar(label='Normalized Convergence Factor (%)')

    plt.show()

# Parameters for the fractal
x_min, x_max = -2, 2
y_min, y_max = -2, 2
resolution = 1000  # Adjust for desired detail and performance
max_iter = 100

# Generate and plot the fractal with cubic convergence visualization
plot_fractal_with_cubic_convergence(x_min, x_max, y_min, y_max, resolution
    , max_iter)
```




```
import numpy as np
```

```
import matplotlib.pyplot as plt
from numba import jit

# Define the custom fractal function with the iterative function
@jit(nopython=True)
def compute_fractal_with_hold_constant(x_min, x_max, y_min, y_max, width, height, max_iter):
    # Create arrays for the real and imaginary parts
    real = np.linspace(x_min, x_max, width)
    imag = np.linspace(y_min, y_max, height)

    # Initialize arrays for the fractal set and divergence
    fractal_set = np.zeros((height, width), dtype=np.int32)
    divergence = np.zeros((height, width), dtype=np.float64)

    for i in range(height):
        for j in range(width):
            c = complex(real[j], imag[i])
            z = complex(0, 0)
            z_prev = complex(0, 0)  # Initialize z_{n-1}
            diverged = False
            for k in range(max_iter):
                # Update z using the iterative function
                z_temp = z  # Store current z to update z_prev later
                z = z**2 - z_prev + c  # Iterative function
                z_prev = z_temp  # Update z_prev for the next iteration

                # Check for convergence or divergence
                if abs(z) >= 4:
                    fractal_set[i, j] = k
                    # Calculate the hold constant (distance from the origin)
                    hold_constant = k * abs(z - z_temp) / abs(z)
                    divergence[i, j] = hold_constant
                    diverged = True
                    break

            if not diverged:
                # Point is in the root set (converges to 0)
                fractal_set[i, j] = max_iter
                # Set the hold constant to 0 for visualization
                divergence[i,j] = 0

    return fractal_set, divergence

def plot_fractal_with_hold_constant(x_min, x_max, y_min, y_max, resolution, max_iter):
    # Compute the fractal set and divergence
    fractal_set, divergence = compute_fractal_with_hold_constant(
        x_min, x_max, y_min, y_max, resolution, resolution, max_iter
    )

    # Create the plot
    plt.figure(figsize=(12, 9))
```

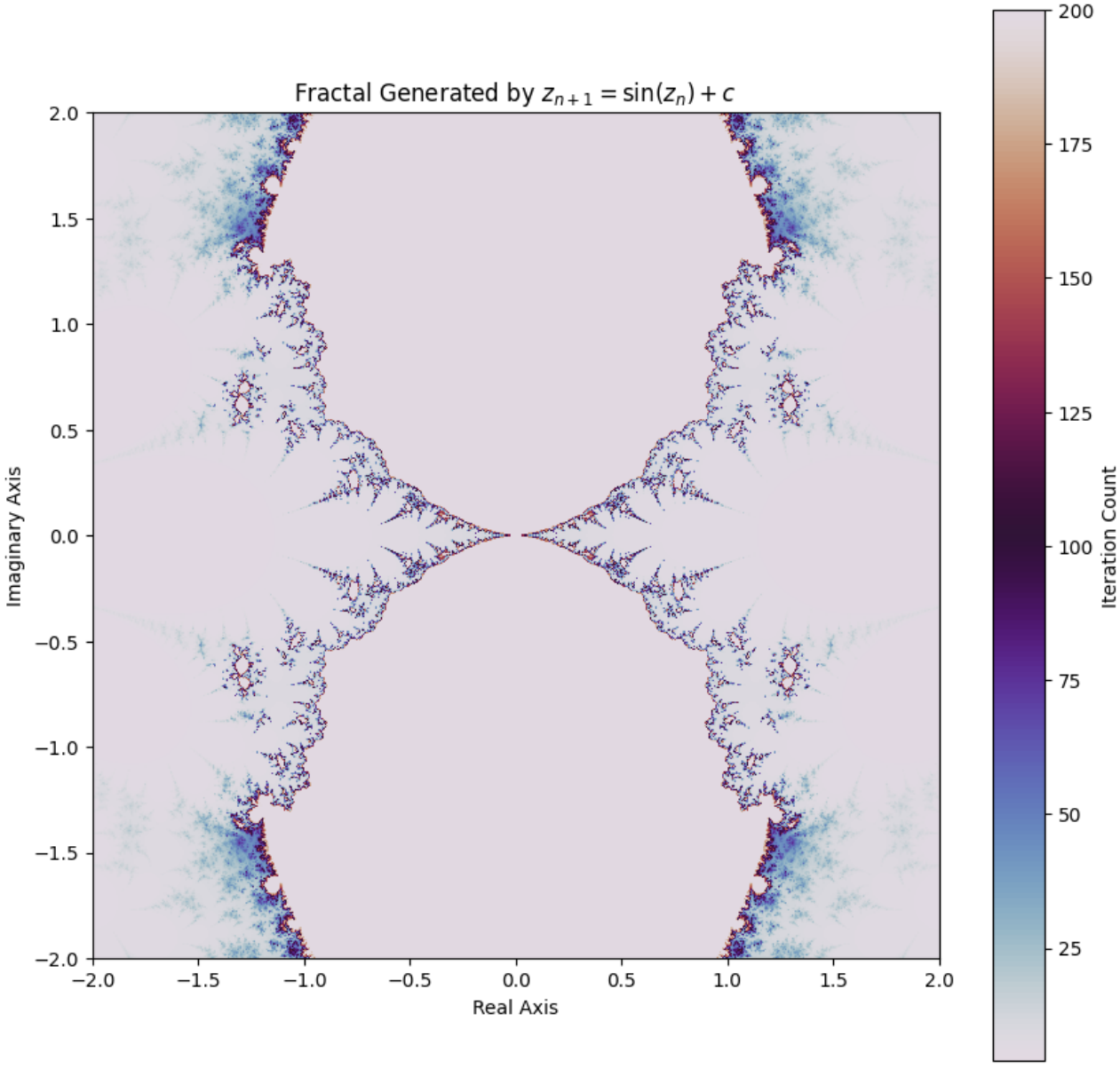
```
    plt.imshow(
        fractal_set.T,
        extent=(x_min, x_max, y_min, y_max),
        cmap='winter',
        interpolation='bilinear',
        origin='lower'
    )
    plt.colorbar(label='Iteration Count')
    plt.title('Fractal Visualization with Hold Constant')
    plt.xlabel('Real')
    plt.ylabel('Imaginary')

    # Overlay the hold constant
    plt.imshow(
        divergence.T,
        extent=(x_min, x_max, y_min, y_max),
        cmap='inferno',
        alpha=0.5,
        interpolation='bilinear',
        origin='lower'
    )
    plt.colorbar(label='Hold Constant')

    plt.show()

# Parameters for the fractal
x_min, x_max = -2, 2
y_min, y_max = -2, 2
resolution = 1000  # Adjust for desired detail and performance
max_iter = 100

# Generate and plot the fractal with hold constant visualization
plot_fractal_with_hold_constant(x_min, x_max, y_min, y_max, resolution,
    max_iter)
```




```python
import numpy as np
import matplotlib.pyplot as plt
from numba import njit
import warnings

# Suppress warnings from invalid operations
warnings.filterwarnings("ignore", category=RuntimeWarning)

# Define the custom fractal function with the iterative function
@njit
def compute_fractal_with_averaged_iterations(x_min, x_max, y_min, y_max,
    width, height, max_iter):
    # Create arrays for the real and imaginary parts
    real = np.linspace(x_min, x_max, width)
    imag = np.linspace(y_min, y_max, height)

    # Initialize arrays for the fractal set and average iteration count
    fractal_set = np.zeros((height, width), dtype=np.int32)
    avg_iter = np.zeros((height, width), dtype=np.float64)

    for i in range(height):
        for j in range(width):
            c = complex(real[j], imag[i])
```

```
            z = complex(0, 0)
            z_prev = complex(0, 0)  # Initialize z_{n-1}
            total_iterations = 0
            for k in range(max_iter):
                # Update z using the iterative function
                z_temp = z  # Store current z to update z_prev later
                z = z**2 - z_prev + c  # Iterative function
                z_prev = z_temp  # Update z_prev for the next iteration

                # Calculate the average iteration count
                total_iterations += k + 1

                # Check for divergence
                if abs(z) >= 4:
                    fractal_set[i, j] = k
                    break

            # Calculate the average iteration count for each point
            avg_iter[i, j] = total_iterations / max_iter

    return fractal_set, avg_iter

def plot_fractal_with_averaged_iterations(x_min, x_max, y_min, y_max, resolution, max_iter):
    # Compute the fractal set and average iteration count
    fractal_set, avg_iter = compute_fractal_with_averaged_iterations(
        x_min, x_max, y_min, y_max, resolution, resolution, max_iter
    )

    # Create the plot
    plt.figure(figsize=(12, 9))
    plt.imshow(
        fractal_set.T,
        extent=(x_min, x_max, y_min, y_max),
        cmap='inferno',
        interpolation='bilinear',
        origin='lower'
    )
    plt.colorbar(label='Iteration Count')
    plt.title('Fractal Visualization with Averaged Iterations')
    plt.xlabel('Real')
    plt.ylabel('Imaginary')

    # Overlay the average iteration count
    plt.imshow(
        avg_iter.T,
        extent=(x_min, x_max, y_min, y_max),
        cmap='viridis',
        alpha=0.5,
        interpolation='bilinear',
        origin='lower'
    )
    plt.colorbar(label='Average Iteration Count')

```

```
    plt.show()

# Parameters for the fractal
x_min, x_max = -2, 2
y_min, y_max = -2, 2
resolution = 1000  # Adjust for desired detail and performance
max_iter = 100

# Generate and plot the fractal with averaged iteration visualization
plot_fractal_with_averaged_iterations(x_min, x_max, y_min, y_max,
    resolution, max_iter)
```




```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
```

```
import time

# Define the fractal computation function
def compute_fractal(width, height, max_iter, x_min, x_max, y_min, y_max):
    # Initialize arrays to store the real and imaginary parts of the
       complex numbers
    real_vals = np.linspace(x_min, x_max, width)
    imag_vals = np.linspace(y_min, y_max, height)
    real, imag = np.meshgrid(real_vals, imag_vals)
    c = real + 1j * imag

    z = np.zeros_like(c, dtype=np.complex128)
    iter_grid = np.zeros_like(c, dtype=np.int32)

    # Iterate to compute the fractal
    for i in range(max_iter):
        mask = np.abs(z) < 1000
        z[mask] = np.sin(z[mask]) + c[mask]
        iter_grid[mask] += 1

    return iter_grid

# Modified functions with exponential component
def f1_modified(theta):
    with np.errstate(divide='ignore', invalid='ignore'):
        result = np.arcsin(np.sin(theta)) + (np.pi / 2) * np.exp(-np.pi /
           (2 * theta))
        result = np.nan_to_num(result, nan=0.0, posinf=0.0, neginf=0.0)
    return result

def f2_modified(theta):
    with np.errstate(divide='ignore', invalid='ignore'):
        result = np.arcsin(np.cos(theta)) + (np.pi / 2) * np.exp(-np.pi /
           (2 * theta))
        result = np.nan_to_num(result, nan=0.0, posinf=0.0, neginf=0.0)
    return result

# Generate points on the unit circle in the first quadrant
num_points = 5000  # Increased resolution
theta = np.linspace(0, np.pi / 2, num_points)
x = np.cos(theta)
y = np.sin(theta)

# Define a range of r values
r_values = np.linspace(0.1, 1.0, 200)  # Increased number of r values

# Create the plot for modified functions
fig, ax = plt.subplots(figsize=(8, 8))

# Plot the unit circle
ax.plot(x, y, 'k-', linewidth=0.5)

# For each r, compute and plot A_r and B_r using modified functions
for r in r_values:
```

```
    A_r_x = []
    A_r_y = []
    B_r_x = []
    B_r_y = []

    for xi, yi in zip(x, y):
        if xi >= 0 and yi >= 0:
            # For A_r with modified f1
            r_xi = r * xi
            if -1 <= r_xi <= 1 and r_xi != 0:
                arcsin_xi = np.arcsin(xi)
                arcsin_r_xi = np.arcsin(r_xi)
                condition_A = arcsin_xi >= f1_modified(arcsin_r_xi)
                if condition_A:
                    A_r_x.append(xi)
                    A_r_y.append(yi)

            # For B_r with modified f2
            r_yi = r * yi
            if -1 <= r_yi <= 1 and r_yi != 0:
                arcsin_yi = np.arcsin(yi)
                arcsin_r_yi = np.arcsin(r_yi)
                condition_B = arcsin_yi >= f2_modified(arcsin_r_yi)
                if condition_B:
                    B_r_x.append(xi)
                    B_r_y.append(yi)

    # Plot the points
    ax.scatter(A_r_x, A_r_y, color='blue', s=0.05, alpha=0.5)
    ax.scatter(B_r_x, B_r_y, color='green', s=0.05, alpha=0.5)

# Customize the plot
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_title('Modified Sets $A_r$ and $B_r$ with Increased Resolution')
ax.axis('equal')
ax.grid(True)
ax.legend(['Unit Circle', '$A_r$ Modified', '$B_r$ Modified'], loc='upper
    right')

# Display the plot
plt.show()

# Perform iterative mapping and visualize the fractal
start_time = time.time()

# Define parameters for the fractal
width = height = 1000  # Increased resolution
max_iter = 200
x_min, x_max = -2, 2
y_min, y_max = -2, 2

# Compute the fractal
divergence_iter = compute_fractal(width, height, max_iter, x_min, x_max,
```

```
    y_min, y_max)

# Create a custom color map
colors = plt.cm.twilight(np.linspace(0, 1, max_iter))
newcmp = ListedColormap(colors)

# Plot the fractal
plt.figure(figsize=(10, 10))
plt.imshow(divergence_iter.T, extent=[x_min, x_max, y_min, y_max], cmap=
    newcmp, origin='lower')
plt.colorbar(label='Iteration Count')
plt.title('Fractal Generated by $z_{n+1} = \sin(z_n) + c$')
plt.xlabel('Real Axis')
plt.ylabel('Imaginary Axis')
plt.show()

end_time = time.time()
print(f"Fractal computation and plotting took {end_time - start_time:.2f}
    seconds.")
```

8 Bibliography

https://www.kea.nu/files/textbooks/humblepy/doingmathwithpython.pdf, D o i n g M a t h with Python, AMIT SAHA